Experience with a Course on Architectures for Software Systems
Part II: Educational Materials

David Garlan, Mary Shaw, José Galmes

August 1994
CMU-CS-94-178

School of Computer Science
Carnegie Mellon University
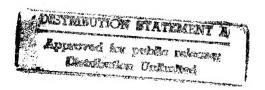Pittsburgh, PA 15213

Abstract

This report contains the materials used by the instructors to teach the course *CS 15-775: Architectures for Software Systems* in Spring 1994 in the School of Computer Science at Carnegie Mellon University. The materials include the lecture slides, questions (with answers) on readings, and homework assignments (with sample solutions).

19950404 152

Computer Science

Experience with a Course on Architectures for Software Systems
Part II: Educational Materials

David Garlan, Mary Shaw, José Galmes

August 1994

CMU-CS-94-178

DTIC

DTIC
SELECTED
APR 0 5 1995
B

Carnegie Mellon

Table of Contents

1. Introduction

It has been two years since we wrote the first part of this technical report as *Experiences with a Course on Architectures for Software Systems: Part I: Course Description* (CMU-CS-92-176, CMU/SEI-92-TR-17, ESC-TR-92-017). Part I contained the rationale behind the course, a course description and a course evaluation. This report contains the educational materials used in the course CS 15-775: Architectures for Software Systems, taught in the Spring 1994 term at Carnegie Mellon University.

The course has now been taught three times to a large number of students. As we anticipated in Part I, we have reorganized the course since its first version. First, the lectures have been factored along themes of architectural idioms. In the first offering of the course the lectures were divided into three major topics: introduction, notations and tools, and formal models and analysis techniques. The result was that each idiom was touched three times. Now all aspects of an architectural idiom are covered in a series of contiguous lectures. Second, the course project now consists primarily of an architectural design for a system with which the students are familiar (preferably one on which they are working), rather than an architectural analysis of an existing system.

The course is now sufficiently mature that a book covering most of the reading materials will be published this fall by Prentice Hall under the title *Software Architecture: Perspectives on an Emerging Discipline*.

1.1 Organization of the document

This document is divided into 3 parts: slides, questions and answers on readings and homework.

Sections 2 through 24 contain the slides used for teaching most of the lectures in the course. You'll notice that some lectures are omitted; they correspond to lectures given by guest lecturers.

Section 25 contains the questions (with answers) for each lecture in the course. There were no questions for a few lectures. Those correspond either to those given by guest lecturers or to class sessions in which there was no lecture (e.g., we used the session for project or homework discussions and presentations).

Sections 26 through 30 contain the homework assignments of the course. This consists of four homework assignments and a course project. For each homework we include the problem description and a sample solution chosen among those turned in by the students.

1.1. Acknowledgments

We would like to thank all the students in the String '94 CS 15-775: Architectures for Software Systems, and especially those who contributed with sample solutions to the homework assignments.

We would also like to thank the guest lecturers for the course: Gregory Abowd, Robert Allen, Robert DeLine, Stuart Feldman, John Ockerbloom, Reid Simmons, Bennett Yee, and Gregory Zelesnik.

Course Information

Garlan & Shaw   February 9, 1994

Class Meetings

Monday and Wednesdays, 10:30-11:50
Wean 3420

Instructors

David Garlan
garlan@cs.cmu.edu
WeH 8020 (x8-5056)
Office Hours: Mon 9:30-10:30
Secretary: Cary Lund, WeH 8106, (x8-3853)

Mary Shaw
mary.shaw@cs.cmu.edu
WeH 8109 (x8-2589)
Office Hours: Wed 3:00-3:30, Th 10:30-11:00
Secretary: Elizabeth Brown, WeH 8107 (x8-3063)

Teaching Assistant

Jose (Pepe) Galmes
galmes@cs.cmu.edu
WeH 4615 (#26) (x8-3826)
Office Hours: Thu 4:00-5:00

Objectives

Architectures for Software Systems aims to teach you how to design, understand, and evaluate systems at an architectural level of abstraction. By the end of the course you should be able to:

- Recognize major architectural styles in existing software systems.
- Describe an architecture accurately.
- Generate architectural alternatives for a problem and choose among them.
- Construct a medium-sized software system that satisfies an architectural specification.
- Use existing definitions and development tools to expedite such tasks.
- Understand the formal definition of a number of architectures and be able to reason precisely about the properties of those architectures.
- Use domain knowledge to specialize an architecture for a particular family of applications.

Version 1.4 of 2/4/94

Organization

Lectures. Class will meet Monday & Wednesday, 10:30-11:50 am, in WeH 3420.

Computing. Some of the assignments will make use of tools that are part of the SCS software engineering environment. You will need an account on an SCS SunOS machine to use these tools. If you are an MSE student you will already have such an account. Other students should see the instructor after the first class to get an application form. There is a document that describes the MSE tool facilities, available from your instructor.

We will be setting up a class afs directory that will contain various templates and documents that will be helpful in completing your homework.

Communication. You will need to read the course bulletin board **cmu.cs.class.cs775** regularly. We welcome e-mail about the course at any time.

Readings. There is no required textbook for this course. Instead, we will use a collection of readings that will be distributed over the course of the semester. There will be a charge to cover the cost of duplications. The readings for the first half of the course (lectures 1-13) will be $20. Elizabeth Brown in WeH 8107 will collect money and distribute the readings. Readings for the second half of the course will be distributed later, probably in the same fashion.

Grading. The course grade will be determined as a combination of four factors:

- **Readings: (25%)** Each lecture will be accompanied by one or more readings, which we expect you to read *before* you come to class. To help you focus your thoughts on the main points of the reading we will assign one or two questions to be answered for each of the reading assignments. Each question should be addressed in less than a page, due before the class for which it is assigned. You can e-mail your solutions to Pepe (galmes@cs.cmu.edu) or turn in hard copy at the beginning of class. Each of these will be graded on a OK/not-OK basis, and will count for about 1% of your grade.

- **Homework Assignments: (40%)** There will be four homework assignments. Each will count 10% of your grade. The first three will be system building exercises. Their purpose is to give you some experience using architectures to design and implement real systems. You will work in groups of three (assigned by us) to carry out each assignment. To help clarify your designs we will hold a brief, un-graded design review for each assignment during class a week before it is due. Groups will take turns presenting their preliminary designs and getting feedback from the class and instructors. The fourth assignment will give you some practice using formal models of software architectures.

- **Project: (25%)** There will be a course project, designed to give you some experience with the architecture of a substantial software system. You design and analyze a new software system from an architectural point of view, document your work, and present the results to the rest of the class. Your grade will depend both on the quality of your design and analysis, and also on its presentation.

- **Instructors' judgment: (10%)**

Important Assignment Dates:

#	Assigned	Discuss	Due	Topic
1	1/26	2/2	2/9	Objects
2	2/9	2/16	2/25	Pipes
3	2/23	3/7	3/14	Events
4	3/14	3/21	4/6	Formal Models
Project	2/21	4/4, 4/6	3/23 progress 4/25 final	Design Task

Schedule

#	Date	Topic	Subtopic and Reading	Assignment
*1	T 1/11	Introduction	Overview and Organization	
2	W 1/12		What is Software Architecture? [GS93 (sec 1-3), Shaw93 (sec 1), PW92]	
3	M 1/17		Classical Module Interconnection Languages [DK76, PN86]	
4	W 1/19	Proc Call	Information Hiding and Objects [PCW85, Boo86]	
5	M 1/24		Modular Decomposition Issues: KWIC [Par72]	A1 distributed
6	W 1/26		Formal Models [Sha85, Spi89, AAG93]	
7	M 1/31	Data Flow	Batch Sequential and Pipeline Systems [Shaw93 (sec 2.intro, 2.1, 3.intro, 3.1, 3.2)]	
8	W 2/2		Tektronix Case Study [GS93 (sec 4.2), GD90]	A1 discussion
9	M 2/7		Implementation Using Unix Pipes [Bac86]	A2 distributed
10	W 2/9		Formal Models for Data Flow [AG92]	A1 due
11	M 2/14	Processes	Communicating Process Architectures [And91]	
12	W 2/16		Formal Models for Processes [Hoa85]	A2 discussion
13	M 2/21	Events	Models of Event Systems [GKN92, GN91]	Project distributed
14	W 2/23		Architecture for Robotics [Sim93]	A2 due on 2/25 A3 distributed
*15	W 3/2		Implementation of Event Systems [Rei90, NGGS93]	
16	M 3/7	Repositories	Blackboard Systems [Nii86]	A3 discussion
17	W 3/9		Databases and Client-Server Systems [GR93, Mul93, C-S93a,b,c]	
18	M 3/14		Evolution of Shared Information Systems [Eco93, Mor93, Shaw93(all)]	A3 due A4 distributed
19	W 3/16	Interpreters	Interpreters and Heterogeneous Systems [GS93 (sec 4.4,4.5, 4.6, 5), Wie92]	
20	M 3/21	Connections	Newer MILs [Per87, GC92, Mak92]	A4 discussion
21	W 3/23		Interface Matching [PA91, Bea92, NHWS91]	Project progress report
*22	M 4/4		Connection Languages [SG93, Sha94a, Sha+94b]	Project discussion
23	W 4/6		Connection Formalisms [AG94]	A4 due Project discussion
24	M 4/11	Specific Architectures	Telephony Architectures [TBD3]	

25	W 4/13		Layered Architectures: OSI [Tan81]	
26	M 4/18		User Interface Architectures [Abowd94]	
27	W 4/20	Design Guidance	Design Assistance [Lan90, ASBD92]	
28	M 4/25	Projects	Final Presentations	Project Due
29	W 4/27		Final Presentations	

* marks classes that follow a holiday

References

[AAG93]   Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proc First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1993.

[AKB94]   Gregory Abowd, Rick Kazman, and Len Bass. *Evaluating the Properties of User Interface Software through Architecture.* Submitted for publication, February 1994.

[AG92]   Robert Allen and David Garlan. Towards Formalized Software Architectures. *Carnegie Mellon University Computer Science Technical Report CMU-CS-92-163*, July 1992.

[AG94]   Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proc Sixteenth International Conference on Software Engineering*, 1994.

[And91]   Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49-90, March 1991.

[ASBD92]   Toru Asada, Roy F. Swonger, Nadine Bounds, and Paul Duerig. The Quantified Design Space: A Tool for the Quantitative Analysis of Designs. *Carnegie Mellon University Computer Science Technical Report CMU-CS-92-213*, November 1992.

[Bac86]   Maurice J. Bach.*The Design of the Unix Operating System*, Chapter 5.12, pp. 111-119. Prentice-Hall Software Series, 1986.

[Bea92]   Brian Beach. Connecting Software Components with Declarative Glue. *Proc. 14th International Conference on Software Engineering*, 1992.

[Boo86]   Grady Booch. Object-Oriented Development. *IEEE Trans. on Software Engineering*, SE-12(2):211-221, February 1986.

[C-S93a]   Maryfran Johnson. Editor's Note. *Client/Server Journal*, 1(1), November 1993, p.3.

[C-S93b]   Jerrold Grochow. Living with the Legacy. *Client/Server Journal*, 1(1), November 1993, p.8.

[C-S93c]   Hugh Ryan. Sticker Shock! *Client/Server Journal*, 1(1), November 1993, pp. 35-38.

[DK76]   Frank DeRemer and Hans H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Trans. on Software Engineering*, SE-2(2):80-86, June 1976.

[Eco93]   The Computer Industry. *The Economist*, Feburary 27th, 1993.

[GC92]   David Gelernter and Nicholas Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97-107, February 1992.

[GD90]   David Garlan and Norman Delisle. Formal Specifications as Reusable Frameworks. In *VDM'90: VDM and Z – Formal Methods in Software Development*. Springer-Verlag, LNCS 428, 1990.

[GKN92]   David Garlan, Gail Kaiser, and David Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer*, 25(6), June 1992.

[GN91]   David Garlan and David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *VDM'91:Formal Software Development Methods*, Springer-Verlag, LNCS 551, 1991.

[GR93]   Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[GS93]   David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds), *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company, 1993.

[Hoa85]   C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall 1985.

[Lan90]   Thomas G. Lane. A Design Space and Design Rules for User Interface Software Architecture. *Carnegie Mellon University Software Engineering Institute Technical Report CMU/SEI-90-TR-23*.

[Mak92]   Victor M. Mak. Connection: An Intercomponent Communication Paradigm for Configurable Distributed Systems. In *Proc. International Workshop on Configurable Distributed Systems*, March 1992.

[Mor93]   Charles R. Morris and Charles H. Ferguson. How Architecture Wins Technology Wars. Harvard Business Review, 71, 2, March-April 1993, pp.86-96.

[Mul93]   Sape Mullender. *Distributed Systems*. Second Edition, Addison-Wesley 1993.

[NHWS91] Gordon. S. Novak, Frederick N. Hill, Man-Lee Wan, and Brian C. Sayrs Negotiated Interfaces for Software Reuse. *IEEE Trans. on Software Engineering*, 18(7):646-653, 1991.

[NGGS93] David Notkin, David Garlan, William G. Griswold, and Kevin Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *Proc. JSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, LNCS 742, November 1993, pp. 489-510.

[Nii86]   H. Penny Nii. Blackboard Systems. *AI Magazine* 7(3):38-53 and 7(4):82-107.

[PA91]   James M. Purtilo and Joanne M. Atlee. Module Reuse by Interface Adaptation. *Software -Practice and Experience*, 21(6):539-556, June 1991.

[Par72]   David L. Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, 15(12):1053-1058, December 1972.

[PCW85]   David L. Parnas, Paul C. Clements, and David M. Weiss. The Modular Structure of Complex Systems. *IEEE Trans. on Software Engineering*, SE-11(3):259-266, 1985.

[Per87]   Dewayne E. Perry. Software Interconnection Models. In *Proc. Ninth International Conference on Software Engineering*, IEEE Computer Society Press, March 1987.

[PN86]   R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4), November 1986, pp. 307-334.

[PW92]   Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, October 1992.

[Rei90]   S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57-66, July 1990.

[SG93]   Mary Shaw and David Garlan. Characteristics of Higher-level Languages for Software Architecture. Unpublished manuscript, 1993.

[Sha85]   Mary Shaw. What Can We Specify? Questions in the Domains of Software Specifications. In *Proc.Third International Workshop on Software Specification and Design*, pp. 214-215. IEEE Computer Society Press, August 1985.

[Sha93]   Mary Shaw. Software Architecture for Shared Information Systems. *Carnegie Mellon University Software Engineering Institute Technical Report CMU/SEI-93-TR-3*, March 1993.

[Sha94a] Mary Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. To appear *Proc Workshop on Studies of Software Design*, Springer-Verlag 1994.

[Sha+94b] Mary Shaw *et al.* Abstractions for Software Architectures and Tools to Support Them. Manuscript.

[Sim93] Reid Simmons. Structured Control for Autonomous Robots. *IEEE Transactions on Robotics and Automation*, 1993.

[Spi89] J. M. Spivey, An Introduction to Z and Formal Specification, *Software Engineering Journal*, Jan 1989.

[Tan81] Andrew S. Tannenbaum. Network Protocols. *ACM Computing Surveys*, 13(4):453-489, December 1981.

[Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38-49, March 1992.

Architectures for
Software Systems

David Garlan

Mary Shaw

with assistance from

Pepe Galmes

Software Architectures ━━━━━━━━━━━━━━━━━━━━━━
1

Why Should You Care?

**Practical concern with the cost and
utility of software**

**Esthetic concern with the clarity of
system structure**
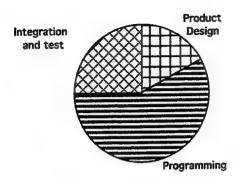
**Some systems with lousy algorithms
work really well**

**Some systems with great algorithms
are really lousy**

Why?

Software Architectures ━━━━━━━━━━━━━━━━━━━━━━
2
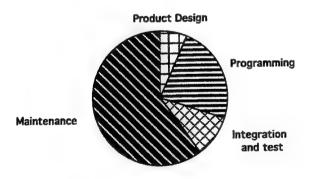
Distribution of Software Development Costs

Integration and test

Product Design

Programming

What's wrong with this picture?

Software Architectures

3
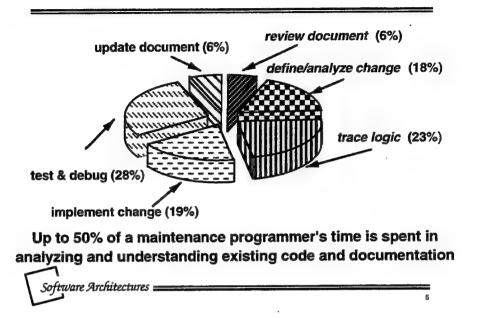
Distribution of Total Software Costs

Product Design

Programming

Maintenance

Integration and test

Software Architectures

4

Allocation of Available Time

update document (6%)

review document (6%)

define/analyze change (18%)

trace logic (23%)

test & debug (28%)

implement change (19%)

Up to 50% of a maintenance programmer's time is spent in analyzing and understanding existing code and documentation

Software Architectures

5

Note on other slides

Here there will be some slides from the Field Guide to American Houses, illustrating various styles and geographical distribution, generally-useful techniques, and design/implementation needed for some specific style.

Software Architectures

6

Analogy to Building Architecture

- **Architectural styles: Colonial, Victorian, Greek Revival, etc.**
  - > Software system organization paradigms
- **Building codes: electrical, structural, etc.**
  - > Formal specifications
- **Special expertise for a given style: balloon frames, slate roofs, etc.**
  - > Domain-specific architectures

Software Architectures

7

Examples of Architecture Diagrams

Examples from literature:

2 all rectangles, all lines alike

3 several box shapes

4 nesting (some substructure shown)

5 adjacent boxes, no lines

Software Architectures

8

Typical Descriptions of Software Architectures

> "Camelot is based on the **client-server model** and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector 87]

> "We have chosen a **distributed, object-oriented approach** to managing information." [Linton 87]

> "The easiest way to make the canonical sequential compiler into a concurrent compiler is to **pipeline** the execution of the compiler phases over a number of processors." [Seshadri 88]

> "The ARC network [follows] the **general network architecture** specified by the ISO in the Open Systems Interconnection Reference Model." [Paulk 85]

Software Architectures

9

Observations about Designers
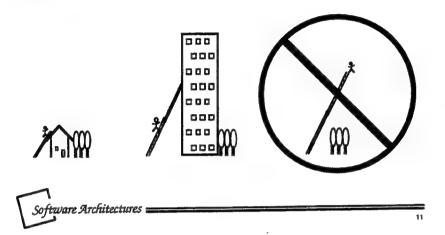
- They freely use informal patterns (idioms).
- They express these in diagrams as well as prose.
- They behave compositionally, building systems from subsystems.
- They tend to think about system structure statically.
- They often select overall organization by default, not by design.

Software Architectures
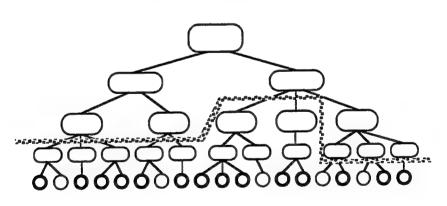
10

Aren't Programming Languages Good Enough?

When orders-of-magnitude improvement are required, new technology may be necessary

Software Architectures

11
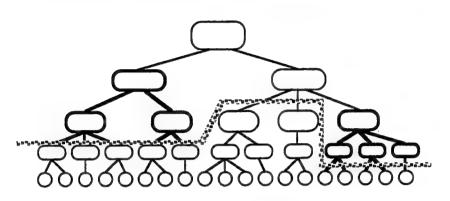
Software Design Levels: Programs

Library Reuse

Software Architectures

12

Software Design Levels: Architecture

Architectural Patterns

Software Architectures

13

Elements of a Complete
Software System

User view of problem	User Model
Software view of problem	Requirement
Modules and connections	Architecture
Algorithms & data structs	Code
Data layouts, memory maps	Executable

Software Architectures

14

Architectural Design Level of Software

- **Deals with the composition of software systems from module-scale elements**
  - **> Gross decomposition of required function**
    - » What are the elements?
    - » How are they connected?
  - **> Assignment of function to design elements**
    - » What patterns of organization are useful?
    - » Which organization fits the application best?
  - **> Scaling and performance**
    - » capacities, flows, balance, schedules
  - **> Selection among design alternatives**
    - » Which implementations of elements will work best?

Software Architectures 15

Architectural Design Task

Different issues for architecture & programs

Architecture	*Programs*
interactions among parts	implementations of parts
structural properties	computational properties
declarative	operational
mostly static	mostly dynamic
system-level performance	algorithmic performance
outside module boundary	Inside module boundary

Software Architectures 16

Elements of a Complete
Software System

User view of problem	**User Model**
Software view of problem	**Requirement**
Modules and connections	**Architecture**
Algorithms & data structs	**Code**
Data layouts, memory maps	**Executable**

Software Architectures

17

Analogy to Building Architecture

- **Architectural styles: Colonial, Victorian, Greek Revival, etc.**
  - > Software system organization paradigms
- **Building codes: electrical, structural, etc.**
  - > Formal specifications
- **Special expertise for a given style: balloon frames, slate roofs, etc.**
  - > Domain-specific architectures

Software Architectures

18

Aren't Programming Languages Good Enough?

When orders-of-magnitude improvement are required, new technology may be necessary

Software Architectures

09068CS2C7

19
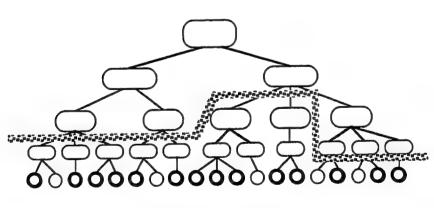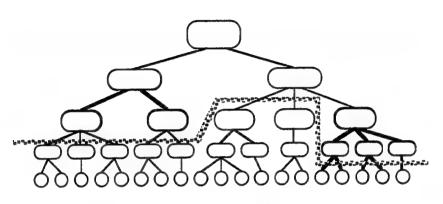
Software Design Levels

Library Reuse

Software Architectures

20

# Software Design Levels

Architectural Patterns

Software Architectures

21

Lecture 2
What Is A Software Architecture Anyhow?

Mary Shaw

Software Architectures

1

Distant Past Software Engineering

1960±5	1970±5
Programming-any-which-way	Programming-in-the-small
Mnemonics, precise use of prose	Simple input-output specifications
Emphasis on small programs	Emphasis on algorithms
Representing structure; symbolic information as well as numeric	Data structures and types
Elementary understanding of control flow	Programs execute once and terminate
State not understood apart from control	Small state space, symbolic or numeric
Program ~ collection of code	Program ~ collection of functions

Software Architectures

2

Past Software Engineering

1970±5	1980±5
Programming-in-the-small	Programming-in-the-large
Simple input-output specifications	Systems with complex specifications
Emphasis on algorithms	Emphasis on interfaces, mgmt., system structure
Data structures and types	Long-lived databases
Programs execute once and terminate	Program assemblies execute continually
Small state space, symbolic or numeric	Large structured state space, symbolic or numeric
Program ~ collection of functions	"Program" ~ collection of components

Software Architectures

3

Historically Useful Strategies

Underlying problem: limited capacity of the human mind to deal with very much complexity at once.

Ways to cope:
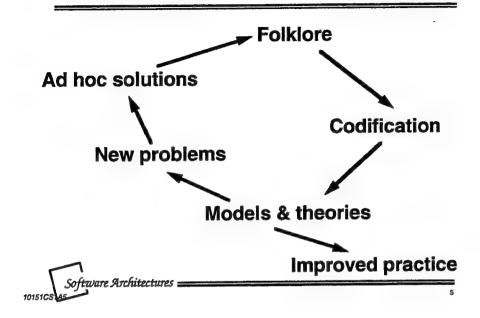
- **Abstraction**
  - > Abstraction is forgetting (suppressing) detail
- **Separation of concerns**
  - > Find parts of a problem that can be solved separately
- **Engineering tools**
  - > Analysis and evaluation models
  - > Common design templates
- **Progressive codification**
  - > Identify, organize, and systematize useful patterns

Software Architectures

4

Good Science Feeds Good Engineering

```
         ┌──────────→ Folklore
Ad hoc solutions            │
    ↑                       ↓
New problems          Codification
    ↑                       │
    │                       ↓
      Models & theories
                            │
                            ↓
                  Improved practice
```

Software Architectures

10151CS\5   5

Abstraction Techniques

Abstraction: a simplified description, or
  specification, of a system that emphasizes
  some details and suppresses others

- A *good* abstraction emphasizes *the right*
  detail.

- Examples:
  > 1950's: mnemonic id's, macros & procedures
  > 1960's: higher-level programming languages
                    algorithms & data structures
  > 1970's: abstract data types & inheritance
  > 1980's: generic definitions, packages

Software Architectures

                                                              6

Experience with a Course on Architectures for Software Systems   3-3

Leverage in Software Development

Requirements

Here a *Miracle* happens

Code

Software Architectures ─────── 7

Leverage in Software Development

Requirements

Architecture

Code

Software Architectures ─────── 8

Elements of Architectural Descriptions

- **The architecture of a system includes**
  - > Components: define the locus of computation
    - » Examples: filters, databases, objects, ADTs
  - > Connectors: define the interactions between components
    - » Examples: procedure call, pipes, event broadcast
- **An architectural style defines a family of architectures constrained by**
  - > Component/connector vocabulary
  - > Topology
  - > Semantic constraints

Software Architectures

9

Common Architectural Idioms

- **Data flow systems**
  Batch sequential   Pipes and filters
- **Call-and-return systems**
  Main program & subroutines   Object-oriented systems
  Hierarchical layers
- **Virtual machines**
  Interpreters   Rule-based systems
- **Independent components**
  Communicating processes   Event systems
- **Data-centered systems (repositories)**
  Databases   Blackboards
- **... and more ...**

Software Architectures

10

Batch Sequential

Data Transformation

tape → Validate → tape → Sort → tape → Update → tape → Report → report

tape

Data Flow

Classical data processing

Software Architectures

11
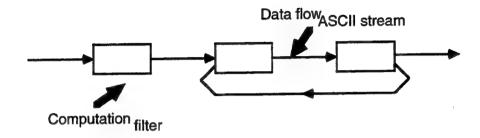
Batch Sequential Systems

- **Processing steps are independent programs**
- **Each step runs to completion before next step starts**
- **Data transmitted as a whole between steps**
- **Typical applications:**
  - > classical data processing
  - > program development

Software Architectures

12

Pipeline

Data flow ASCII stream

Computation filter
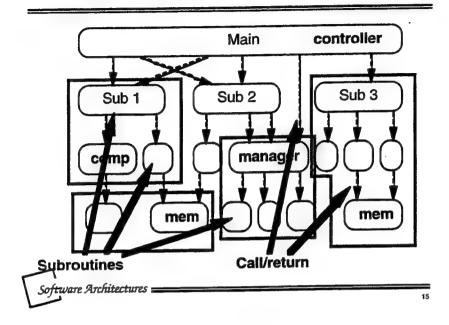
Software Architectures

Pipes and Filters

- **Filter**
  - > Incrementally transform some amount of the data at inputs to data at outputs
    - » Stream-to-stream transformations
  - > Use little local context in processing stream
  - > Preserve no state between instantiations
- **Pipe**
  - > Move data from a filter output to a filter input
  - > Pipes form data transmission graphs
- **Overall Computation**
  - > Run pipes and filters (non-deterministically) until no more computations are possible.

Software Architectures

14

Main Program/Subroutine Pattern

Main   controller

Sub 1   Sub 2   Sub 3

comp   manager

mem   mem

Subroutines   Call/return

Software Architectures

15
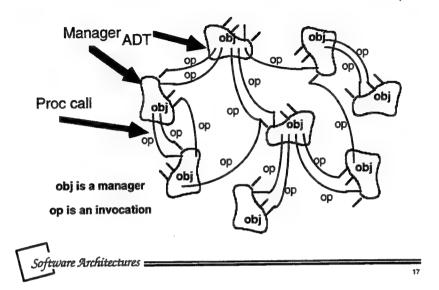
Main Program and Subroutines

- **Hierarchical decomposition:**
  > Based on definition-use relationship

- **Single thread of control:**
  > Supported directly by programming languages

- **Subsystem structure implicit:**
  > Subroutines typically aggregated into modules

- **Hierarchical reasoning:**
  > Correctness of a subroutine depends on the correctness of the subroutines it calls

Software Architectures
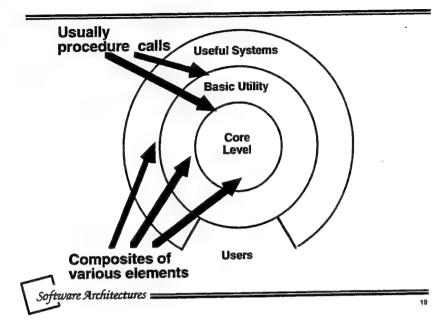
16

Data Abstraction or Object-Oriented

Manager ADT

Proc call

obj is a manager

op is an invocation

Software Architectures

17

Object Architectures

- **Encapsulation:**
  > Restrict access to certain information
- **Inheritance:**
  > Share one definition of shared functionality
- **Dynamic binding:**
  > Determine actual operation to call at runtime
- **Management of many objects:**
  > Provide structure on large set of definitions
- **Reuse and maintenance:**
  > Exploit encapsulation and locality

Software Architectures

18

Layered Pattern

Usually
procedure calls

Useful Systems

Basic Utility

Core
Level

Composites of
various elements

Users

Software Architectures

19
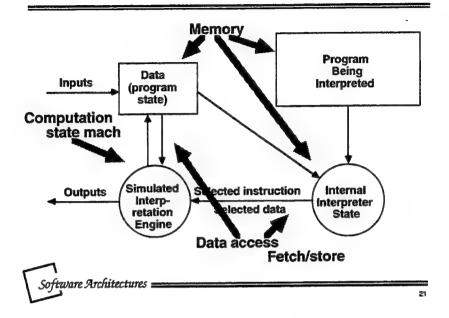
Layered Patterns

- **Each layer provides certain facilities**
  - > hides part of lower layer
  - > provides well-defined interfaces
- **Serves various functions**
  - > kernels: provide core capability, often as set of procedures
  - > shells, virtual machines: support for portability
- **Various scoping regimes**
  - > Opaque versus translucent layers

Software Architectures

20

Interpreter

Memory

| Inputs → | Data (program state) | | Program Being Interpreted |

Computation state mach

Simulated Interp-retation Engine

Internal Interpreter State

Outputs ←

Selected instruction

Selected data

Data access

Fetch/store

Software Architectures

21
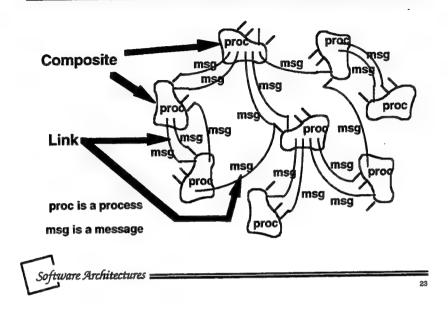
Interpreters

- **Execution engine simulated in software**
- **Data:**
  - > representation of program being interpreted
  - > data (program state) of prog. being interpreted
  - > internal state of interpreter
- **Control resides in "execution cycle" of interpreter**
  - > but simulated control flow in interpreted program resides in internal interpreter state
- **Syntax-driven design**

Software Architectures
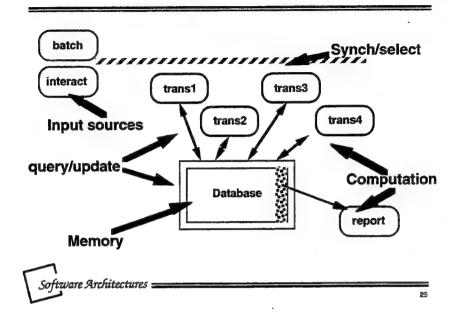
22

Communicating Processes

Composite

Link

proc is a process

msg is a message

Software Architectures

23

Communicating Processes

- **Components: independent processes**
  > typically implemented as separate tasks
- **Connectors:message passing**
  > point-to-point
  > asynchronous and synchronous
  > RPC and other protocols can be layered on top

Software Architectures

24

Repository: Database

- batch
- interact
- **Input sources**
- **Synch/select**
- trans1
- trans3
- trans2
- trans4
- query/update
- **Database**
- **Computation**
- **Memory**
- report

Software Architectures
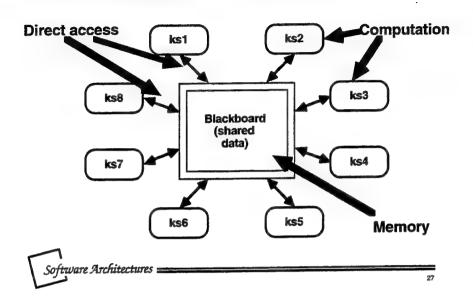
Classical Databases

- **Central data repository**
    - > Schemas designed specifically for application
- **Independent operators**
    - > Operations on database implemented independently, one per transaction type
    - > Interact with database by queries and updates
- **Control**
    - > Transaction stream drives operation
    - > Operations selected on basis of transaction type

Software Architectures

Repository (Blackboard) · · · ·

Direct access   ks1   ks2   Computation

ks8   **Blackboard (shared data)**   ks3

ks7   ks4

ks6   ks5   Memory

Software Architectures

27

The Blackboard Model

- ## Knowledge Sources
  - > World and domain knowledge partitioned into separate, independent computations
  - > Respond to changes in blackboard

- ## Blackboard Data Structure
  - > Entire state of problem solution
  - > Hierarchical, nonhomogeneous
  - > Only means by which knowledge sources interact to yield solution

- ## Control
  - > In model, knowledge sources self-activating

Software Architectures

28

Comparison of System Patterns ...

System Model	Components	Connections	Control Struct
Pipeline			
stream -> stream	filters (local processing)	data flow ASCII streams	data flow
Data abstraction (object-oriented)			
localized state maint	servers (ADTs, objs)	procedure call	decentralized, single thread
Repository			
central database	1 memory N processes	direct access or proc call	internal or external
Interpreter			
virtual machine	state mach, two memories	fetch, store	input-driven

Software Architectures

29

Common Architectural Idioms

- **Data flow systems**
  - Batch sequential   Pipes and filters
- **Call-and-return systems**
  - Main program & subroutines   Object-oriented systems
  - Hierarchical layers
- **Virtual machines**
  - Interpreters   Rule-based systems
- **Independent components**
  - Communicating processes   Event systems
- **Data-centered systems (repositories)**
  - Databases   Blackboards
- **... and more ...**

Software Architectures

30

Experience with a Course on Architectures for Software Systems

3-15

Important Ideas

- *Common patterns* for system structure
  - > pure type forms, allowing variation
  - > Identifiable types of *subsystems* and *interactions*

- *Decomposition and heterogeneity:*
  - > patterns also describe subsystem structure
  - > subsystem pattern ≠ system pattern;

- *Independence:*
  - > system patterns and subsystem functions don't depend on application

- *Fit to problem:*
  - > problem characteristics guide choice of structure

Software Architectures ===

31

Architectural Design Level of Software

- **Deals with the composition of software systems from module-scale elements**
  - > Gross decomposition of required function
    - » What are the elements?
    - » How are they connected?
  - > Assignment of function to design elements
    - » What patterns of organization are useful?
    - » Which organization fits the application best?
  - > Scaling and performance
    - » capacities, flows, balance, schedules
  - > Selection among design alternatives
    - » Which implementations of elements will work best?

Software Architectures ===

32

Architectural Design Task

Different issues for architecture & programs

Architecture	Programs
interactions among parts	implementations of parts
structural properties	computational properties
declarative	operational
mostly static	mostly dynamic
system-level performance	algorithmic performance
outside module boundary	Inside module boundary

Software Architectures

33

Leverage in Software Development

Requirements

Here a *Miracle* happens

Code

Software Architectures

34

Leverage in Software Development

Requirements

Architecture

Code

Batch Sequential

Data Transformation

tape → **Validate** → tape → **Sort** → tape → **Update** → tape → **Report** → report

tape

Data Flow

Classical data processing

Pipes and Filters

Data flow ASCII stream

Computation filter

Software Architectures

06088CS1015

37

Main Program/Subroutine Pattern

Main controller

Sub 1   Sub 2   Sub 3

comp   manager

mem   mem

Subroutines   Call/return

Software Architectures

38

Data Abstraction or Object-Oriented

Manager ADT

obj

op

obj

op

op

op

op

op

Proc call

obj

op

op

op

op

obj

op

op

op

op

obj

obj

obj is a manager

op is an invocation

obj

op

op

Software Architectures

06088CS

39

Layered Pattern

Usually procedure calls

Useful Systems

Basic Utility

Core Level

Composites of various elements

Users

Software Architectures

40

Interpreter Pattern

Communicating Processes

proc is a process

msg is a message

Experience with a Course on Architectures for Software Systems

3-21

Repository: Database

Repository Pattern (Blackboard)
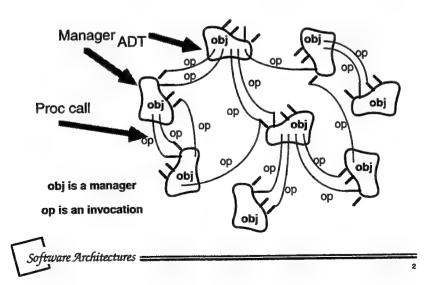
Lecture 4
Information Hiding, Abstract Data Types, Objects

Mary Shaw

Software Architectures

1

Data Abstraction or Object-Oriented Pattern

Manager ADT

Proc call

obj is a manager

op is an invocation

Software Architectures

2

Overview

- **Objective: What does an ADT/O-O approach buy you as compared to conventional block-structured programming?**
- **Outline:**
  - > What problems need ^solved? *(to be)*
  - > Technical difficulties
  - > Design difficulties
  - > Encapsulation
  - > Structure for large definitions
  - > Inheritance

Software Architectures

3

Problems Facing Software Developers of 1972

- **Vulnerability of global variables**
  - > Classical block structure creates sharing
- **Inadvertent disclosure of structure**
  - > Exact location of field, linear vs linked representation
- **Rippling design decisions**
  - > One change may affect many modules
- **Dispersion of code related to a single decision**
  - > It may be hard to locate everything affected
- **Families of related design**
  - > Related definitions de-localize decisions

Software Architectures

4

Elements of solution

* **Technical support**
  > Data access and definitions
  > Locality of definition (encapsulation)
* **Design support**
  > Separation of concerns (encapsulation)
  > Imposition of structure
  > Related definitions

Software Architectures

5

Data Access and Definition Problems

* **Access to internal representation:**
  > *Vulnerability:* Visible representations can be manipulated in unexpected, undesired, and dangerous ways
* **Forced distribution of knowledge:**
  > *Non-Uniform Referent:* Syntax may reveal structure
* **Coupling:**
  > *Instance independence:* When multiple instances of a given struct. are active, they must remain independent
* **Families of definitions:**
  > *Dynamic binding:* If shared definitions involve type variants, function variants must be chosen at runtime

Software Architectures

6

Uniform Referent Problem

- **Non-uniform referents:**
  Vector1[index] := Vector2[index]
  Record1.field1 := Record2.field2
  Set(FieldAdr(A, PseudoVar1), FieldVal(B, PseudoVar2))

- **Uniform referents:**
  Vector1(index) := Vector2(index)
  Record1(field1) := Record2(field2)
  PseudoVar1(A) := PseudoVar2(B)

- **To avoid propagating knowledge of repres.:**
  Use uniform syntax for access functions
  Allow type-specific overloading of :=

Software Architectures

7

Iteration Problem

- **Non-uniform referents:**
  for i := 1 step 1 until N do somefunc(V[i])

  or

  p := V;
  while p~= nil do { p := p.next; somefunc(q.val)}

- **For each structured type:**
  > define how an iteration proceeds uniformly through the structure
  > allow this to be connected to syntax of loops

- **Generators (a.k.a. iterators):**
  > forall x in V do somefunc(x)

Software Architectures

8

Locality Problems

- **Access to internal representation:**
    - > *Nonlocality:* **If the way something is used depends on how it's imp'd., you must find all uses to make a change**
- **Forced distribution of knowledge:**
    - > *Non-localized operations:* **Some operations may implicitly reveal representation: iteration, input/output, ...**
- **Coupling:**
    - > *Global data:* **Fnal. decomp. often exposes critical data**
    - > *Shared assumptions:* **create implicit interdependence**
- **Families of definitions:**
    - > *Similar definitions:* **Shared function should be defined once only**

Software Architectures

9

Encapsulation

- **Parnas: Hide secrets (not just representations)**
- **Booch: Object's behavior is characterized by actions that it suffers and that it requires**
- **Practically speaking:**
    - > Object has state and operations, but also has responsibility for the integrity of its state
    - > Object is known by its interface
    - > Object is probably instantiated from a template
    - > Object has operations to access and alter stateand perhaps generator
    - > There are different kinds of objects (e.g, actor, agent, server)

Software Architectures

10

Abstract Data Types

- **Late 1960's: Good programmers shared an intuition: if you get the data structures right, the rest of the program is much simpler.**
- **Abstract Data Type Research of 1970's:**
  - > *Structure* (representation bundled with operators)
  - > *Specifications* (abstract models, algebraic axioms)
  - > *Language* (modules, scope, user-defined types)
  - > *Integrity constraints* (invariants of data structures)
  - > *Rules for combining types* (declarations)
  - > *Information hiding* (protect properties not in specs)
- **Routine practice now part of o-o discipline**

Software Architectures ================================ 11

Objects

- **Booch intro implies essence of object-ness is i/f: operations it provides and requires**
  - > But this is too general -- it includes subroutines
- **More generally,**
  - > Allow use in terms of specifications alone   *this*
  - > "Hide" representation; use other objects   *part*
  - > Maintain state   *like*
  - > Provide subroutines for actions   *ADTs*
  - > Sustain conceptual coherence
  - > Instantiate from template
  - > Support definitional inheritance

Software Architectures ================================ 12

Remark on Cruise Control Example

- **Example assumes that the only choices are functional and o-o and that the data flow diagram is "the right" place to begin**
  - > Data flow diagrams aren't the only way to start
  - > Functional decompositions often have hidden dependencies
  - > Object decompositions allow aliasing to be created
- **In fact, this problem can be addressed as a control loop, which leads to a quite different structure**

Software Architectures

13

Managing Large Object Sets

- **Pure o-o design leads to large flat systems with many objects**
  - > Same old problems can reappear
  - > Hundreds of modules => hard to find things
  - > Need a way to impose structure
- **Need additional structure and discipline**
- **Structuring options**
  - > Layers (which are not necessarily objects)
  - > Supplemental index
  - > Hierarchical decomposition: big objects and little objects (not much discussed)

Software Architectures

14
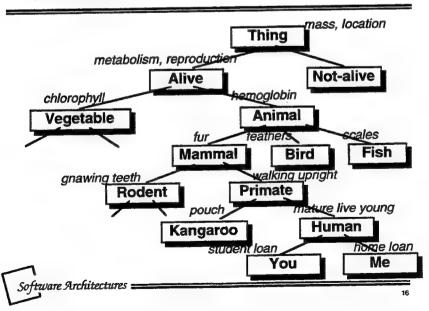
Management of Many Objects ...

- **Parnas: Build definition hierarchy (independent of "uses" structure or call tree)**
- **For A7E, structure is**
  - > **Hardware-hiding (hardware/software interfaces)**
  - > **Behavior-hiding (implications of changeable requirements)**
  - > **Software decisions (design decisions based on math, physics, programming considerations)**
- **Note that many things are hidden besides representations**

Software Architectures ══════════════════════════════
15

Inheritance

Software Architectures ══════════════════════════════
16

Reuse and Maintenance
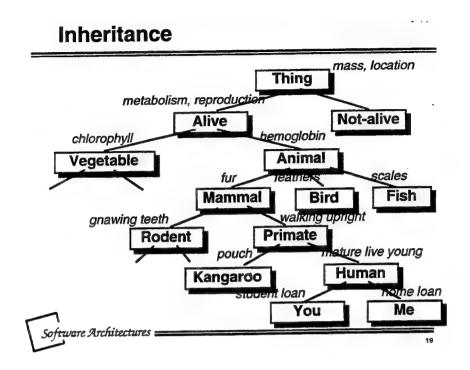
- Object organization decreases system coupling
- This should reduce propagation of changes
- It should increase ease of understanding system and therefore reduce cost of maintenance
- It should make individual objects more independent of system, therefore more likely to be reusable
  > But this is over-rated
- It should become possible to build and re-use *frameworks* (architectures for particular kinds of systems) by standardizing interfaces

Software Architectures

17

Elements of Object Architectures

- **Encapsulation: Restrict access to certain information**
- **Inheritance: Share one definition of shared functionality**
- **Management of many objects: Provide structure on large set of definitions**
- **Reuse and maintenance: Exploit encapsulation and locality to increase productivity**

  Note that the object architecture closely resembles the object programming style

Software Architectures

18

Inheritance

Software Architectures

19

Lecture 5
Modular Decomposition Issues:
KWIC

David Garlan

Software Architectures ══════════════════════════
1

Purpose of This Lecture

- Discuss "On the Criteria To Be Used in Decomposing Systems into Modules", Parnas (1972).

- Detailed example of use of Information Hiding and Abstract Data Types as an architectural style.

- Explore advantages and disadvantages of this style.

- Motivate the implementation assignments for this course.

Software Architectures ══════════════════════════
2

What is a Modularization?

- **Common view:**
  - > A *module* is a piece of code.
  - > *Modularization* decomposes the code of a system into smaller pieces.

- **Parnas view:**
  - > A *module* defines a unit of responsibility.
  - > *Modularization* decomposes the overall responsibility (for satisfying requirements) into smaller pieces.

- **Hence Parnas is concerned with *interfaces* -- or what must be prescribed before implementation can begin.**

Software Architectures
3

Why Modularize?

- ***Managerial:*** Partition the overall development effort (divide and conquer).

- ***Evolution:*** Decouple parts of a system so that changes to one part are isolated from changes to other parts.

- ***Understandability:*** Permit system to be understood as composition of mind-sized chunks.

Software Architectures
4

Key Word In Context

Problem Description:

"The KWIC index system accepts and ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.

Any line may be 'circularly shifted' by repeatedly removing the first word and appending it at the end of the line.

The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."

On the Criteria for Decomposing Systems into Modules. David Parnas. *CACM*, 1972

Software Architectures

5

KWIC: Key Word In Context

- **Inputs: *Sequence of lines***

    ```
    Pipes and Filters
    Architectures for Software Systems
    ```

- **Outputs: *Sequence of lines, circularly shifted and alphabetized***

    ```
    and Filters Pipes
    Architectures for Software Systems
    Filters Pipes and
    for Software Systems Architectures
    Pipes and Filters
    Software Systems Architectures for
    Systems Architectures for Software
    ```

Software Architectures

6

Design Considerations

- **Change in Algorithm**
  - > Eg., batch vs incremental
- **Change in Data Representation**
  - > Eg., line storage, explicit vs implicit shifts
- **Change in Function**
  - > Eg., eliminate lines starting with trivial words
- **Performance**
  - > Eg., space and time
- **Reuse**
  - > Eg., sorting

Software Architectures ═══════════════
7

Solution 1: Design

- **Decompose the overall processing into a sequence of processing steps.**
  - > Read lines; Make shifts; Alphabetize; Print results
- **Each step transforms the data completely.**
- **Intermediate data stored in shared memory.**
  - > Arrays of characters with indexes
  - > Relies on sequential processing

Software Architectures ═══════════════
8

Solution 1: Modularization

- **Module 1: Input**
  - > Reads data lines and stores them in "core".
  - > Storage format: 4 chars/machine word; array of pointers to start of each line.

- **Module 2: Circular Shift**
  - > Called after Input is done.
  - > Reads line storage to produce new array of pairs:
    - (index of 1st char of each circular shift, index of original line)

- **Module 3: Alphabetize**
  - > Called after Circular Shift.
  - > Reads the two arrays and produces new index.

Software Architectures

9
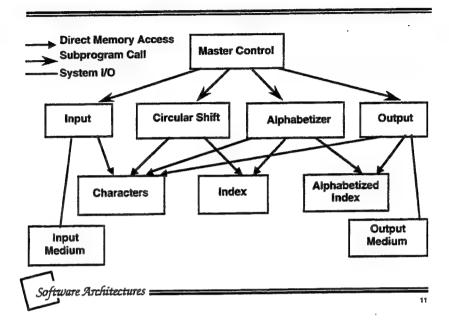
Solution 1: Modularization (2)

- **Module 4: Output**
  - > Called after alphabetization and prints nicely formatted output of shifts
  - > Reads arrays produced by Modules 1 & 3

- **Module 5: Master Control**
  - > Handles sequencing of other modules
  - > Handles errors

Software Architectures

10

Architecture of Solution 1

> **Direct Memory Access**
> **Subprogram Call**
> **System I/O**
>
> Master Control
> Input — Circular Shift — Alphabetizer — Output
> Characters — Index — Alphabetized Index
> Input Medium — Output Medium

Software Architectures

11

Properties of Solution 1

- **Batch sequential processing.**
- **Uses shared data to get good performance.**
- **Processing phases handled by control module.**
  - > So has some characteristics of main program - subroutine organization.
- **Shared data structures exposed as inter-module knowledge.**
  - > Design of these structures must be worked out before work can begin on those modules.

Software Architectures

12

Solution 2: Design

- **Maintain same flow of control, but**
- **Organize solution around set of data managers (objects):**
  - > for initial lines
  - > shifted lines
  - > alphabetized lines
- **Each manager:**
  - > handles the representation of the data
  - > provides procedural interface for accessing the data

Software Architectures

13

Solution 2: Modularization
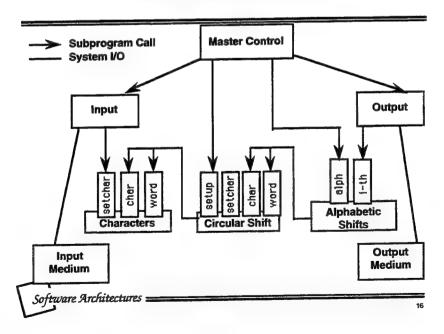
- **Module 1: Line storage**
  - > Manages lines and characters; procedural interface
  - > Storage format: not specified at this point
- **Module 2: Input**
  - > Reads data lines and stores using "Line Storage"
- **Module 3: Circular Shift**
  - > Provides access functions to characters in circular shifts
  - > Requires CSSETUP as initialization after Input is done

Software Architectures

14

Solution 2: Modularization (2)

- **Module 4: Alphabetize**
  - > Provides index of circular shift
  - > ALPH called to initialize after Circular Shift
- **Module 5: Output**
  - > Prints formatted output of shifted lines
- **Module 6: Master Control**
  - > Handles sequencing of other modules

Software Architectures

15

Architecture of Solution 2

Software Architectures

16

Properties of Solution 2

- **Module interfaces are abstract**
  - > hide data representations
    - could be array + indices, as before
    - or lines could be stored explicitly
  - > hide internal algorithm used to process that data
    - could be lazy or eager evaluation
  - > require users to follow a protocol for correct use
    - initialization
    - error handling
- **Allows work to begin on modules before data representations are designed.**
- **Could result in same executable code as first solution.**

Software Architectures

17

Comparisons

- **Change in Algorithm**
  - > Solution 1: batch algorithm wired into
  - > Solution 2: permits several alternatives

- **Change in Data Representation**
  - > Solution 1: Data formats understood by many modules
  - > Solution 2: Data representation hidden

- **Change in Function**
  - > Solution 1: Easy if add a new phase of processing
  - > Solution 2: Modularization doesn't give particular help

Software Architectures

18

Comparisons (2)

- **Performance**
  - > Solution 1: Good
  - > Solution 2: Probably not as good, but might be
- **Reuse**
  - > Solution 1: Poor since tied to particular data formats
  - > Solution 2: Better

Software Architectures

19

Principles of Information Hiding

- **Hide the right secrets**
  - > A "right secret" is a design decision that is likely to change
- **Data representations are one such secret**
  - > A data manager provides a set of data accessing procedures that allow it to control
    - » integrity of the data
    - » actual representation
  - > Then we get Abstract Data Types (or Objects)

Software Architectures

20

Principles of Information Hiding (cont.)

- **Can also hide HW/SW infrastructure**
  - > cf., A7 paper
  - > A virtual machine provides an abstraction of the actual hardware/software functionality
  - > Then get a layered system
- **Can hide nature of concurrent access to facilities provided by a module**
  - > Then get a client-server system

Software Architectures

Some Distinctions

- **Hierarchy versus Information Hiding**
  - > Hierarchy: no circular dependencies
  - > Can have a hierarchical system without info hiding
  - > Can have a system that uses info hiding but is not hierarchical
- **ADTs versus Information Hiding**
  - > Can hide other things than data representation

Software Architectures

Some Distinctions (2)

- **Uses versus Calls**
  - > In a later paper Parnas makes the distinction between *uses* and *calls*
    - » If A *uses* B then A's correctness depends on B's correctness
    - » If a *calls* B then A may or may not depend on B's correctness
  - > Calls but not uses: Module A calls "Done" when finished
  - > Uses but not calls: Alphabetize depends on Input

Software Architectures

23

Lecture 6
Formal Models

David Garlan

Software Architectures ═══════════════════════════

1

The Purpose of This Lecture

- **Explain why formal models can provide insight into software architecture**
- **Provide an introduction to Z**
  - The Mathematical Basis of Z
  - A Simple Example
  - The Schema Calculus
- **Clarify the use of Z for understanding architectural style**

Software Architectures ═══════════════════════════

2

Outline

* **Why Formal Models?**
  > What is a formal model of a software architecture?
  > Why are formal models useful?
  > What can we formalize?

* **The Z Specification Language**
  > The mathematical basis of Z
  > A simple Example
  > The schema calculus

Software Architectures ═══════════════════════
3

Why Formal Models of Software Architecture?

* **A formal model is a mathematical abstraction.**

* **Benefits:**
  > Abstraction: What is the essence of an architectural style?
  > Precision: How can we make informal use more scientific?
  > Analysis: What can we predict about an architecture?
  > Codification: Can we provide standard reference models for architecture?
  > Comparison: How are different architectures related?
  > Automation: What kinds of tool support can we develop?

Software Architectures ═══════════════════════
4

What Can We Formalize?

- <u>Structure</u>: How is a system organized?
- <u>Compatibility</u>: When is a system properly composed?
- <u>Function</u>: What does the system compute?
- <u>Resource usage</u> :How fast/big is it?
- <u>Invariants</u>: What are the "load-bearing walls"?
- <u>Specializations</u>: How do specific systems constrain more general models?

Software Architectures ======================

5

In This Course

- We will see:
  - > How to characterize state spaces and transitions
  - > An industrial case study of a formal model of a product family
  - > A formal reference architecture for pipe/filter systems
  - > A formal treatment of event systems and several common variations
  - > Techniques for formalizing concurrent systems

Software Architectures ======================

6

Why Specify using Z?

- **Allows you to use simple mathematics to document software designs as:**
  - > a client/implementor interface
  - > a technique for reasoning about designs
  - > a method for establishing correctness of implementations

Software Architectures

Formal Underpinnings

- **Mathematical types describe system state in problem-oriented terms.**
  - > Sets
  - > Relations
  - > Functions
  - > Sequences
- **First order predicate logic specifies collections of states and operations by saying "what" not "how".**

Software Architectures

Structure of Z Specifications

- **Schemas describe:**
  - > what states a system can occupy
  - > what operations can happen
  - > relationships between parts of a complex system

Software Architectures ===================================

9

Use of Prose

- **In good Z specifications schemas are presented with informal text which:**

- **motivates the formal descriptions**

- **relates the model to reality**

- **documents requirements**

Software Architectures ===================================

10

The Mathematics of Z: Sets

- In Z sets are typed. That is, the elements are drawn from a common set.
- Examples:

  {1, 3, 5, 7, ... }

  {red, green, blue, blue, green}

  {Joe, Neil, Marco, ... }

  { yes, no }

- No:

  {red, 1, 2, 3 }

Software Architectures

Sets (2)

- A type is just a set.
- One type is predefined -- the set of integers: Z
- Other sets are introduced as <u>given sets</u>

  [Date]   [Person]   [Book, Author]

- Or defined using various <u>set constructors</u>

Software Architectures

Set Comprehension

- **One way of constructing new sets is to define a set using propositions and predicates.**
- **Examples:**

  $N == \{n: Z \mid n \geq 0\}$ (note double =)

  small == $\{x: N \mid x > 3 \wedge x < 6\}$

  mse-fm == $\{p:$ **Person** $\mid p \in$ 17-712 $\wedge$ $p \in$ **mse** $\}$

  squares == $\{x: N \mid (\exists y: N \bullet x = y)\}$

- **This is called <u>set comprehension</u>.**

Software Architectures

Set Comprehension (2)

- **The most general form of a set comprehension is**

  {Declarations | Predicate • Expression}

- **Examples:**

  $\{x: N \mid x > 3 \wedge x < 6 \bullet x\} = \{4, 5\} = \{16, 25\}$

  $\{x: N \bullet 2x\}$

  $\{x: N \mid (\exists y: N \bullet x = y) \bullet 3y\}$

Software Architectures

Variables

- **All variables are typed**
- **Examples:**
  - x: N
  - n: Prime
- **Global variables are defined as follows:**

$$\begin{array}{|l} max: N \\ \underline{min: N} \\ min \geq 3 \end{array}$$

Software Architectures ======================================
15

Enumerated Types

- **Enumerated types can be described as follows:**
  - Status ::= Yes | No
  - Color :: Red | Blue | Green | Yellow
- **This is short hand for**
  - [Status]

$$\begin{array}{|l} \underline{Yes: Status \; ; \; No : Status} \\ Yes \neq No \\ x \in Status \Rightarrow (x = Yes \lor x = No) \end{array}$$

Software Architectures ======================================
16

Power Sets

- **The set of all subsets of S: <u>set of</u> S, or $\underline{P}$ S**
- **Usually referred to as the <u>power set</u> of S**
    - > **<u>Examples</u>:**

        $P\{1, 2, 3\} = \{ \emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

        $P\,N = \{ \emptyset, \{1\}, ...\}$
- **If x : $P\,N$, then x is a set of integers.**

Software Architectures

Tuples

- **A <u>tuple</u> is an ordered pair.**
    - > **<u>Examples</u>:**

        $(2, 3) \neq (3, 2)$

        $S = \{ (2, red), (5, blue), (3, red) \}$
- **The set of tuples constructed from two sets is called a <u>Cartesian Product,</u> or just <u>Product</u> of those sets.**

    $N \times N$

    $S: P\,(N \times Color)$

Software Architectures

Relations

- A <u>relation</u> is a set of tuples.
  - \> <u>Examples:</u>

    A == { (1,1), (1,2), (2,2) }

    B == { (2, red), (5, blue), (3, red) }

    C == { (David, Jun 1), (Mary, Aug 2), (Bill, Feb 5) }

- The set of all relations over sets S, T is indicated by $S \leftrightarrow T$.
  - \> <u>Examples:</u>

    A: $N \leftrightarrow N$

    B: $N \leftrightarrow$ Color

    C: Person $\leftrightarrow$ Date

Software Architectures

19

Relations (2)

- The <u>domain</u> of a relation is the set of first elements.

- The <u>range</u> of a relation is the set of second elements.
  - \> <u>Examples:</u>

    A == { (1,1), (1,2), (2,2) }

    dom A = { 1, 2 } and ran A = { 1, 2 }

    B == { (2, red), (5, blue), (3, red) }

    dom B = { 2,3,5 } and ran B = { red, blue }

    C == { (David, Jun 1), {Mary, Aug 2}, {Bill, Feb 5) }

    dom C = { David, Mary, Bill} and ran C = { Jun 1, Aug 2, Feb 5 }

Software Architectures

20

Functions

- **A function is a relation such that no two distinct tuples contain the same first element.**
  - > **Examples:**
    - B == { (2, red), (5, blue), (3, red) }
    - C == { (David, Jun 1}, {Mary, Aug 2}, {Bill, Feb 5) }
  - > **Not a function:**
    - A == { (1,1), (1,2), (2,2) }

Software Architectures ===================================

Functions (2)

- **The set of all functions between sets S and T is indicated $S \rightarrow T$.**
- **The set of partial functions is indicated $S \rightarrow T$**
- **So**
  - > B: $N \rightarrow$ Color
- **and**
  - > C: Name $\rightarrow$ Date

Software Architectures ===================================

Summary: Set Constructors

- 1. Given Sets
- 2. Enumerated Types
- 3. Power set constructor: $\mathbb{P}$
- 4. Tuples (Cartesian Product)
- 5. Relations
- 6. Functions and Partial functions

Software Architectures

23

A Simple Example [1]

- A small database for recording people's names and birthdays.
- The system should allow us to:
  > add new people to the database
  > lookup the birthday of a person
  > find the names of people with birthdays on a given day

1. From: J.M. Spivey, *The Z Notation*, 1989

Software Architectures

24

The State Space

- **The state of a system is described by a schema.**

```
┌─SchemaName ──────────────
 declarations
 ─────────────────────────
 state invariant
└──────────────────────────
```

BirthdayBook

[Name, Date]

```
┌─BirthdayBook ──────────
 known: P NAME
 birthday: NAME ↦ DATE
 ─────────────────────────
 known = dom birthday
└──────────────────────────
```

Example

known = { David, Judy, Robert }

birthday = { David $\longmapsto$ 24-June,
              Judy $\longmapsto$ 26-August,
              Robert$\longmapsto$ 8-July}

The invariant is satisfied:

known= *dom* birthday

Software Architectures

27

Observation

- **There is:**
  - > No limit on the number of entries
  - > No implied order of entries
  - > No restriction on format
- **But there is a precise statement that:**
  - > Each person has only one birthday
  - > Two people may share a birthday
  - > Some people may not be in the database

Software Architectures

28

Operations

```
___ AddBirthday _____
  Δ BirthdayBook

  name?: NAME

  date?: DATE
|_____
```

Observations:
- state before the operation
- state after the operation
- inputs and outputs

Software Architectures
29

Operations (2)

```
__AddBirthday _____
  Δ BirthdayBook
                        ↘
  name?: NAME
                    _____
  date?: DATE      | known, known'
|_____  | birthday, birthday'
                   |_____
                   | known = dom birthday
                   | known' = dom birthday'
                   |_____
```

Software Architectures
30

Add Birthday

```
┌─AddBirthday ───────────────────
│ Δ BirthdayBook
│ name?: NAME         ┌─────────────────┐
│                     │  pre-condition  │
│ date?: DATE         └─────────────────┘
├──────────────────      ┌──────────────────┐
│                        │  post-condition  │
│ name? ∉ known          └──────────────────┘
│ birthday' = birthday ∪
│    { name? ⟼ date? }
│
└────────────────────────────────
```

Software Architectures ════════════════════════════
                                                      31

Derived Components

- **The invariant**
  known = *dom* birthday

- **allows us to calculate <u>known</u> from <u>birthday</u>.**

- **It is a <u>derived component.</u>**

- **Laws:**
  $dom\,(f \cup g) = dom\,f \cup dom\,g$
  $dom\,\{\,a \longmapsto b\,\} = \{\,a\,\}$

Software Architectures ════════════════════════════
                                                      32

Reasoning About Specifications

known' = *dom* birthday'

   = *dom* (birthday ∪ {name? ↦ date?})

   = *dom* birthday ∪ *dom* {name? ↦ date?}

   = *dom* birthday ∪ {name?}

   = known ∪ {name?}

Software Architectures

Find Birthday

┌─FindBirthday ─────────

Ξ BirthdayBook

name?: NAME

date!: DATE

─────────

name? ∈ known

date! = birthday(name?)

Δ BirthdayBook
─────────
known = known'

birthday = birthday'

Software Architectures

Remind

```
┌─Remind ──────────────────────
│ Ξ BirthdayBook
│ date?: DATE
│ names!: P DATE
├──────────────────────────────
│ names! =
│     {n: known | birthday(n) = date?}
└──────────────────────────────
  n ∈ names! ↔
      n ∈ known ∧ birthday(n) = date?
```

Summary

- **State space**

    +

- **Fragile operations**
- **That is, if the the pre-condition of any operation is violated the system may:**
  - > ignore the operation
  - > crash
  - > break down later

Error Handling

Modify each operation to return a result.

REPORT ::=
    ok | already-known | not-known

┌─Success ─────────────
│ result!: REPORT
├──────────────────────
│ result! = ok
└──────────────────────

Software Architectures ════════════════════════
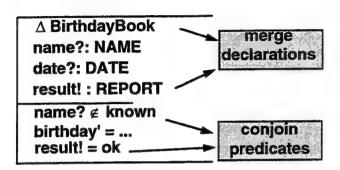
Successful Operations

┌─Success ─────────────
│ result!: REPORT
├──────────────────────
│ result! = ok
└──────────────────────

AddBirthday ∧ Success ──────▶
    Add a birthday, if possible,
    and report ok

Software Architectures ════════════════════════

Add Birthday- revised

```
┌─ Δ BirthdayBook ──────────
│  name?: NAME
│  date?: DATE
│  result! : REPORT
├───────────────────────────
│  name? ∉ known
│  birthday' = ...
│  result! = ok
```

merge declarations

conjoin predicates

Detecting Errors

```
┌─ AlreadyKnown ──────────
│  Ξ BirthdayBook
│  name?: NAME
│  result! : REPORT
├──────────────────────────
│  name? ∈ known
│  result! = already-known
```

If name? is already known don't change
anything and report already-known

Combining the Parts

RAddBirthday $\hat{=}$
   (AddBirthday $\wedge$ Success)
      $\vee$ AlreadyKnown

The Other Operations

Similarly:

RFindBirthday $\hat{=}$
   (FindBirthday $\wedge$ Success)
      $\vee$ NotKnown

RRemind $\hat{=}$
   Remind $\wedge$ Success

already robust

Advantages of Approach

- **Separation of concerns:**
  - > consider each idea separately
- **Focus of understanding:**
  - > mind-sized chunks
- **Modularity:**
  - > reuse pieces

Software Architectures
43

Observation

- **It is possible to combine specifications using ∧ and ∨ using the Schema Calculus.....**

- **....<u>even though you can't combine programs!</u>**

Software Architectures
44

Other Uses

- **Separate:**
  - \> single entity (e.g., process, file, record)
  - \> and its place in the larger system
  - \> different views of the same system
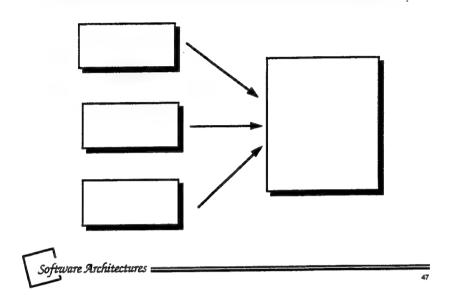 - \> system functions
  - \> and access control

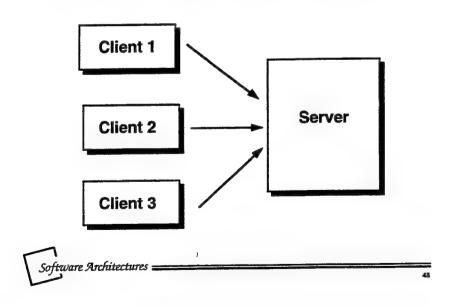*Software Architectures* ====================
45

Summary

- **Z is a simple mathematical framework in which to:**
  - \> describe systems abstractly yet precisely
  - \> compose a system out of small pieces
  - \> use old specifications to build new specifications
  - \> reason about properties of a system
  - \> relate views of a system
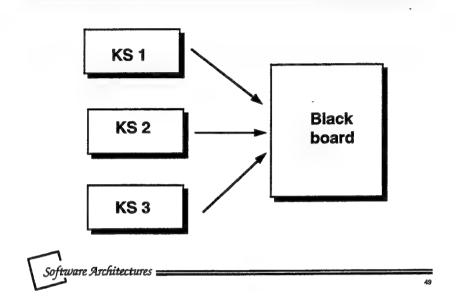
Software Architectures ====================
46

Part III
What is an Architectural Style?
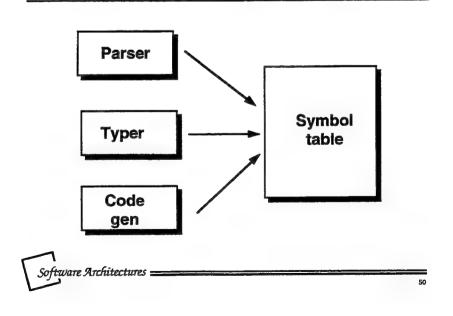
Software Architectures

47

What is an Architectural Style?

Client 1

Client 2

Client 3

Server

Software Architectures

48

What is an Architectural Style?

KS 1 → Black board
KS 2 → Black board
KS 3 → Black board

Software Architectures

49

What is an Architectural Style?

Parser → Symbol table
Typer → Symbol table
Code gen → Symbol table
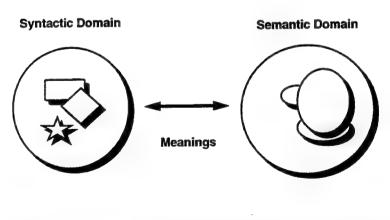
Software Architectures

50

Elements of Architectural Style

- **A family of interoperable components and connectors**
  - > Constraints on vocabulary of comp. & conn.
  - > Example: clients and servers
- **Patterns of system composition**
  - > Constraints on topologies of components and connectors
  - > Example: pipeline
- **Conventions about the meaning of architectural descriptions**
  - > Constraints on semantics
  - > Example: lines mean pipes, boxes mean filters

Software Architectures

51

The Specification Enterprise

Syntactic Domain   Semantic Domain

Meanings

Software Architectures

52

Making Style Precise

- **Syntactic domain**
  - > components (the boxes)
  - > connectors (the lines)
  - > configurations (the topologies)
- **Semantic domain**
  - > sets, tuples, etc.
- **A bridge between the two**
  - > M_{comp}: Components $\mapsto$ ...
  - > M_{conn}: Connectors $\mapsto$ ...
  - > M_{conf}: Configurations $\mapsto$ ...

Software Architectures ========================

53

Why Bother?

- **By looking at inverse map can detect implicit syntactic constraints**
  - > syntactic elements without well-defined semantic meanings should be excluded
  - > example: "broadcast pipes"
- **Allows us to make comparisons between styles**
  - > Pipe and Filters have hierarchical closure property
  - > some Event Systems don't
- **Provides basis for formal analysis**

Software Architectures ========================

54

Lecture 7

Data Flow Architectures: Batch Sequential and Pipeline Systems

Mary Shaw

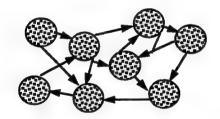*Software Architectures*

1

Objectives

- **Characterize data flow systems**
- **Show limitations**
- **Distinguish between batch sequential and pipeline systems**
- **Introduce systems integration**
- **Mention other kinds of data flow systems**
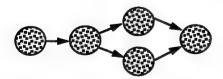
Software Architectures

2

Data Flow Systems

- **A data flow system is one in which**
  - > availability of data controls computation
  - > the structure of the design is dominated by orderly motion of data from process to process
  - > the pattern of data flow is explicit

- **In a pure data flow system, there is no other interaction between processes**
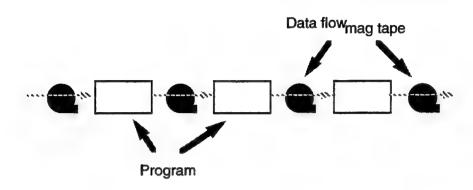
Software Architectures

Kinds of Data Flow Systems

In general, data can flow in arbitrary patterns

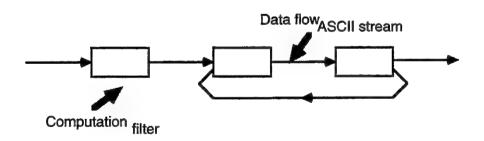Here we are primarily interested in nearly-linear data flow systems

Software Architectures

Batch Sequential Pattern

Data flow mag tape

Program

Software Architectures

5

Pipeline Pattern

Data flow ASCII stream

Computation filter

Software Architectures

6

Systems Integration

- *Systems integration is a problem-solving activity that entails harnessing and coordinating the power and capabilities of information technology in ways tailored to meet a customer's well-defined needs.*
  - \> Includes both organizational and technical issues.

- It's hard:
  - \> large, untidy problems
  - \> incomplete, imprecise, inconsistent requirements
  - \> saddled with old systems that can't be replaced

- Focus here on technical issues.

Software Architectures

7

Your **System**

is

My **Component**

Software Architectures

8

Technical Issues in Integration

- **Architecture**

  System organization: kinds of components, kinds of interactions, patterns of overall organization

- **Connectivity**

  Mechanisms for moving data between systems and initiating action in other systems

- **Semantics**

  Representations, conceptual consistency, semantic models, means for handing inconsistencies

- **Interaction**

  Granularity, user interface, interoperability

Software Architectures

9
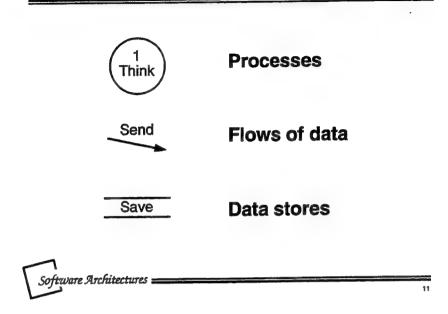
Database Management

- **Business data processing**
  - > Historically dominated by database updates
  - > Discrete transactions of predetermined type; periodic reports; special handling of bad requests

- **Historical base: batch sequential**
  - > Mainframes and magtapes
  - > Manual block scheduling

- **Technology pressure: on-line access**
  - > Queries are relatively easy
  - > On-line updates require shift from pure batch to interactive processing

Software Architectures

10

Yourdon Data Flow Diagrams

(1 Think) **Processes**

Send → **Flows of data**

| Save | **Data stores**

Batch Sequential Data Processing

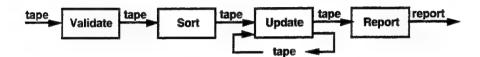- Laurence J. Best. Application Architecture: Modern Large-Scale Information Processing. Wiley 1990.
  - > Bubble diagram of batch sequential form (fig 4-2 p.29)
  - > "calls" relation for program/subprogram structure for update in previous figure (fig 15-2 p.150)

Batch Sequential Architecture

====================

- **Processing steps are independent programs**
- **Each step runs to completion before next step starts**

Software Architectures ================================

Interactive Data Processing

====================
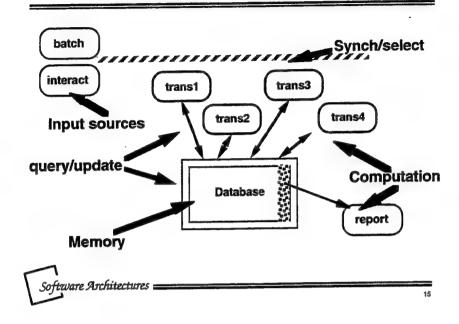
- **Laurence J. Best. Application Architecture: Modern Large-Scale Information Processing. Wiley 1990.**
  - > interactive view of system (fig 8-1 p.81)
  - > interactive view; fine structure of update function (fig 15-5 p.158)

Software Architectures ================================

Repository Architecture

Software Architectures ===== 15

Computer Aided Software Engineering

- **Initially just translation from source to object code: compiler, library, linker, make**
- **Grew to include design record, documentation, analysis, configuration control, incrementality**
- **Integration demanded for 20 years, but not here yet**

Software Architectures ===== 16

CASE vs DBMS

- **As compared to databases, CASE has:**
  - > more types of data
  - > fewer instances of each type
  - > slower query rates
  - > larger, more complex, less discrete information
  - > but *not* shorter lifetime

Software Architectures

17
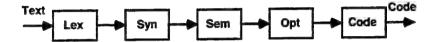
Software Tools with Scripts

- **Earliest tools were independent programs**
  - > Often their output appeared only on paper

- **Next generation shared only files**
  - > Files in universally readable format, but effective sharing limited by lack of information about representation
  - > Tools sequenced with scripts: JCL, simple shell scripts

- **Essentially batch sequential**

Software Architectures

18
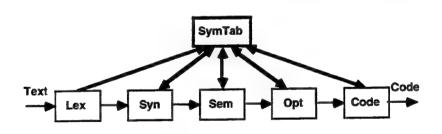
Example: Canonical Compiler

Text → **Lex** → **Syn** → **Sem** → **Opt** → **Code** → Code

Software Architectures

19
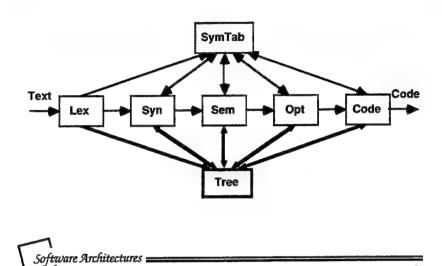
Canonical Compiler: Troublesome Details

SymTab

Text → **Lex** → **Syn** → **Sem** → **Opt** → **Code** → Code
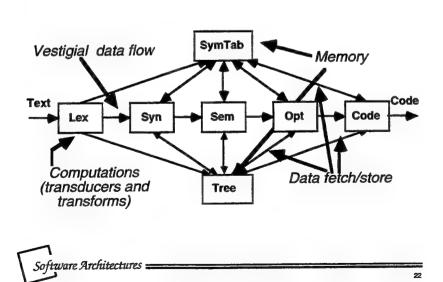
Pipeline?
No, Batch Sequential
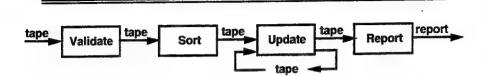
Software Architectures

20

Example: Modern Canonical Compiler

Software Architectures

21

Example: Modern Canonical Compiler

Software Architectures

22
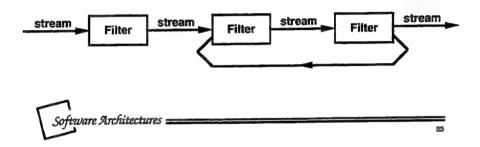
Batch Sequential

Pipe-and-Filter (UNIX)

Software Architectures

Batch Sequential vs Pipe & Filter (UNIX)

Both
Decompose task into fixed sequence of computations
Interact only through data passed from one to another

Batch Sequential	Pipe/Filter
Coarse-grained	Fine-grained, incremental
High latency (real-time is hard)	Results start immediately
Random access to input ok	Processing localized in input
No concurrency	Feedback loops possible
Non-interactive	Often interactive, but awkward
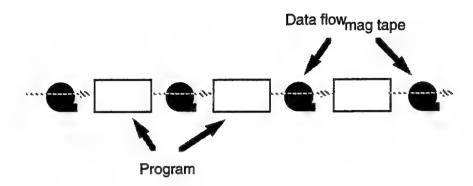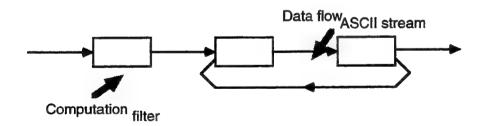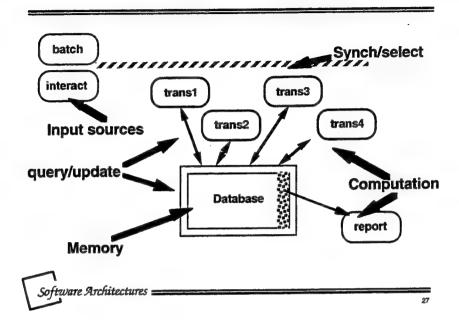
Software Architectures

Batch Sequential Pattern

Data flow mag tape

Program

Software Architectures
25

Pipeline Pattern

Data flow ASCII stream

Computation filter

Software Architectures
06088CST016
26

Repository Architecture

Software Architectures
27

Example: Modern Canonical Compiler

Software Architectures
28
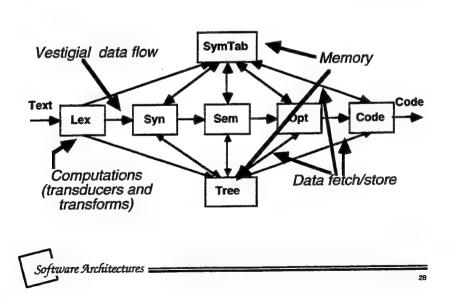
Lecture 8
A Case Study in Pipe/Filter Systems:
The Tektronix Experience

David Garlan

Software Architectures

1

Outline

- The Problem
- How We Addressed It
- The Role of Formalism
- Lessons Learned

Software Architectures

2

The Problem

- **Increasing complexity of instrumentation systems.**
- **Separate development cultures.**
- **Build-from-scratch methods.**
- **Inflexible products.**
- **Excessive time-to-market (~ 4-5 years).**
- **More serious bugs due to concurrency.**

Software Architectures

3
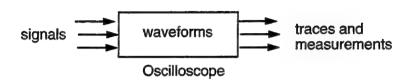
The Challenge

- **Allow reuse between product divisions**
- **Build next generation instrumentation systems**
- **Support better interactive response to user changes**
- **Multiple hardware platforms for same user interface**
- **Multiple user interfaces for same platform (vertical markets)**

Software Architectures

4
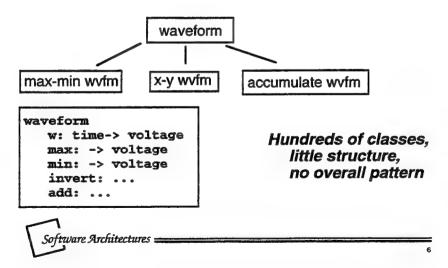
The Arena: Oscilloscopes

signals → **waveforms** → traces and measurements

Oscilloscope

Software Architectures

5
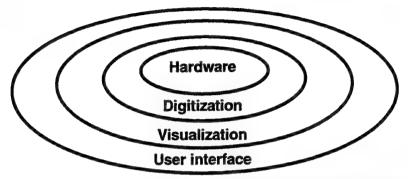
Oscilloscope: O-O Approach

First attempt was an Object-Oriented Decomposition

waveform

| max-min wvfm | x-y wvfm | accumulate wvfm |

```
waveform
   w: time-> voltage
   max: -> voltage
   min: -> voltage
   invert: ...
   add: ...
```

*Hundreds of classes,
little structure,
no overall pattern*

Software Architectures

6

Oscilloscope: Layered Approach

Second attempt was a Layered Architecture
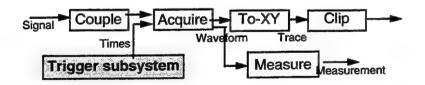
Boundaries of abstraction not realistic

Oscilloscope: Pipe-Filter Approach

Third attempt was a Pipe-Filter Architecture
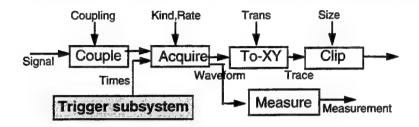
Better, but not clear how to model user input.

Oscilloscope:
Extended Pipe-Filter Approach

Pipe-Filter Architecture with Parameterized Filters.
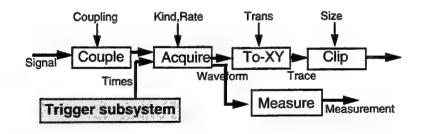
Elegant model, but not directly useful to implementors.

Software Architectures

9

Oscilloscope: Solution

**Pipe-Filter Architecture with
Parameterized Filters and Colored Pipes**

Elegant model, and implementable.

Software Architectures

10

Experience with a Course on Architectures for Software Systems

Results

- Models used as basis for next generation of oscilloscope products.
- Led to highly successful framework, in which time-to-market has been cut to about 6 months-1 year.
- High reliability of products.
- Flexibility of user interface.
- Led to new frameworks beyond oscilloscopes.
- Major thrust of research/development collaborations.

Software Architectures

11

What is a Waveform?

- Ans 1: a sampled signal.
- Ans 2: a 5K array of 8-bit samples.
- Ans 3: a function from time to volts
- Ans 4: a partial function from time to volts
- Ans 5: a relation between time and volts
- Ans 6: a bag of time-volt pairs

Software Architectures

12

Functional View
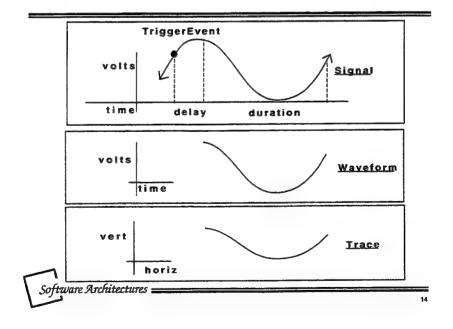
Traces = Oscilloscope (Signals)

But:
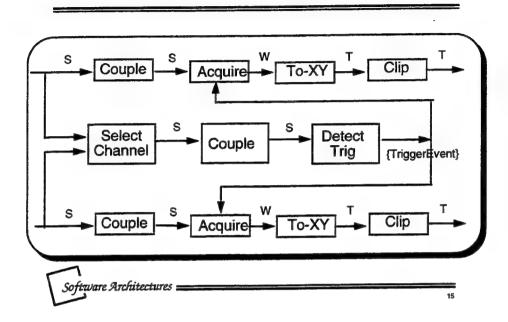
1. How to handle user input?
2. How can you decompose it
   into manageable pieces?

Software Architectures

13

Signals, Waveforms, Traces

Software Architectures

14

The Whole System

Software Architectures   15

Oscilloscope

┌─**Oscilloscope**────────────────────────────
s1, s2: Signal
cp1, cp2: ChannelParameters
tp: TriggerParameters
ts1, ts2: seq Trace
├──────────────────────────────────────
∀ t: ran ts1 •
   ∃ trig: TriggerConfiguration tp (s1, s2) •
      t = ChannelConfiguration cp1 trig s1
∀ t: ran ts2 • ...
└──────────────────────────────────────

Software Architectures   16

Other Architectural Models

- **Connection framework (colored P/F)**
  - > Shared data
  - > Triggered filters
  - > Variable rates
  - > Flexible use of inputs
- **User interface architecture**
  - > Flow of control from front panel to internals
  - > Menu hierarchies
  - > Limited real-estate

Software Architectures

17

User Interface Problem
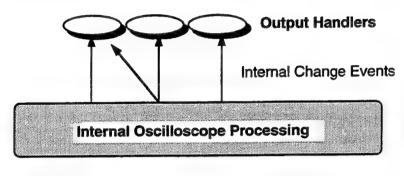
Recall Goals:

- **Multiple hardware platforms for same user interface**
- **Multiple user interfaces for same platform**
- **How to separate user interface from application?**

Software Architectures

18

Case Study (continued)

Output: application announces events

Output Handlers

Internal Change Events

Internal Oscilloscope Processing

Software Architectures

19

Case Study (continued)

Input: user generates events

Input Event Dispatcher

Input Handlers

Reconfiguration Commands

Abstract Interface

Internal Oscilloscope Processing

Software Architectures

20
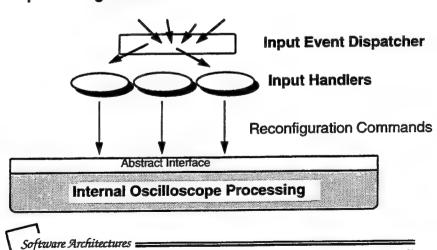
Some Morals

- **Success requires**
  - > Domain experts and system builders
  - > Expertise in abstraction and formal models
  - > Willingness to abandon old design patterns
  - > Willingness to reject inappropriate architectures
- **Tools**
  - > Not needed during conceptualization
  - > Essential during development
- **Management**
  - > Must keep hands off the process initially
  - > Must help enforce standards later

Software Architectures

21

Role of Formalism

- **What was formalized**
  - > User level model (extended P/F)
  - > Connection framework (colored P/F)
  - > User interface architecture
- **Benefits of formalism**
  - > Motivated and constrained architectures
  - > Conceptual prototyping
  - > Communication medium
- **Non-benefits**
  - > Correctness, completeness
  - > Adoption of formal methods within Tektronix

Software Architectures

22

Model for Industrial Research

Development
Effort

| Individual product development |
| Product-line frameworks (company assets) |
| Off-the-shelf components (OS, tools, compiler, ...) |

Software Architectures

23

Lecture 10
Pipe/Filter Systems
(A Formal Approach)

Robert Allen

Software Architectures

1
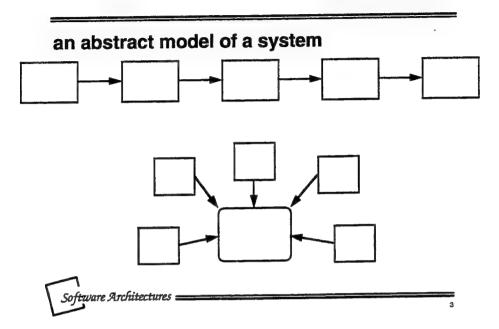
Overview
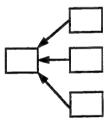
- **Architectural Description (Revisited)**
- **The Need for Formalization**
- **Example: PF**
- **PF Formalized**
- **Using the Formalism**

Software Architectures

2

What is a Software Architecture?

an abstract model of a system

Are pictures enough?

No.
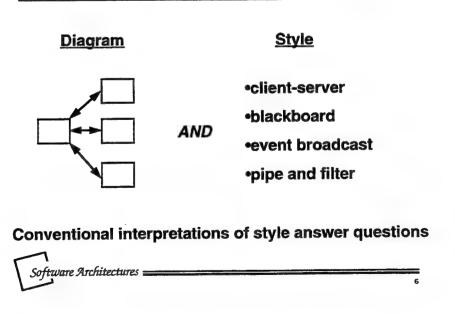
- What happens in the boxes?
- Are the boxes similar in behavior?
- What control/data relationships hold?
- What is the overall behavior?
- Does the diagram make sense?

Making descriptions precise

use restricted syntax

☐   *means*   abstract data type

⟶   *means*   remote procedure call

•unambiguous
•implementable
•limited expressiveness

Software Architectures

5

Architectural style as description

<u>Diagram</u>   <u>Style</u>

AND

•client-server

•blackboard

•event broadcast

•pipe and filter

Conventional interpretations of style answer questions

Software Architectures

6

Problem of Informal Definition · · ·

- **Informal usage presents a number of difficulties**
  - > Lack of shared understanding
  - > Difficulty of communication
  - > Insufficient analytic leverage
  - > Inability to select among architectures

Software Architectures ══════════════════

7

The Need for Formalization

- **A formal approach offers a number of benefits**
  - > Precise definition of paradigm
  - > Medium of communication
  - > Improved analysis of systems
  - > Comparison with other architectures

Software Architectures ══════════════════

8

Experience with a Course on Architectures for Software Systems

Issues Raised by Formalization

• We need to consider a number of issues when developing a formal model
  > What ambiguity should we resolve, and how?
  > What is an appropriate level of abstraction?
  > What systems belong in the architectural family?
  > What properties will we want to analyze?

Software Architectures

9

Pipes and Filters (informal)

• A <u>Filter</u> transforms streams of data.
  > reads data from input ports
  > writes data to output ports

• A <u>Pipe</u> controls the flow of data through the system.
  > links an output port to an input port
  > indicates the path that data will take
  > carries out data transmission

Software Architectures

10

Formalizing PF (overview)

- **Elements:**
  - \> Filter
  - \> Pipe
  - \> System
- **Aspects:**
  - \> Description
  - \> State
  - \> Computation

Software Architectures

Some Preliminary Definitions

- **[FILTER, PORT]**
- **[DATA, FSTATE]**
- **Port_State == PORT** £ **seq DATA**
- **Partial_Port_State == PORT** £ **seq DATA**

Software Architectures

Schema Filter

```
┌─ Filter ──────────────────────────────────────────────┐
│ filter_id  ℙ FILTER                                    │
│ in_ports,out_ports : ℙ PORT                            │
│ alphabets: Port £ ℙ DATA                               │
│ states: ℙ FSTATE                                       │
│ start: FSTATE                                          │
│ transitions: (FSTATE ¡(Partial_Port_State))            │
│                   ↦ (FSTATE ¡(Partial_Port_State))     │
├────────────────────────────────────────────────────────┤
│ start    states                                         │
│ in_ports    out_ports = fl                              │
│ dom alphabets = in_ports    out_ports                   │
│ ((s1,input_obs),(s2,output_gen)    transitions          │
│     s1    states   s2    states                         │
│       dom input_obs = in_ports    dom output_gen = out_ports │
│       ( p: in_ports (ran(input_obs(p)) † alphabets(p)   │
│       ( p: out_ports (ran(output_gen(p)) †alphabets(p)  │
└──────────────────────────────────────────────────────────┘
```

Schema Filter-State

```
┌─ Filter_State ─────────────────────────────────────────┐
│ f : FILTER                                              │
│ internal_state : FSTATE                                 │
│ input_state, output-state : Port_State                  │
├──────────────────────────────────────────────────────────┤
│ internal_state    f.states                              │
│ dom input_state = f.in_ports                            │
│ dom output_state = f.out_ports                          │
│   p: f.in_ports · ran(input_state(p)) † f.alphabets(p)  │
│   p: f.out_ports · ran(output_state(p)) † f.alphabets(p)│
└──────────────────────────────────────────────────────────┘
```

Schema Filter-Compute

```
┌─ Filter_Compute ──────────────────────────────────────────
│ Δ Filter_State
├───────────────────────────────────────────────────────────
│ f = f '
│ « in_consumed,out_gen : Partial_Port_State ·
│     ((internal_state,in_consumed), (internal_state ,'out_gen)   f.transitions
│       ( p: f.in_ports ·
│         input_state(p) = in_consumed(p)  ^ input_state (p))
│       ( p: f.out_ports ·
│         output_state(p) ^  out_gen(p) = output_state (p))
└───────────────────────────────────────────────────────────
```

Software Architectures

15

Schema Pipe

```
┌─ Pipe ────────────────────────────────────────────────────
│ source_filter,sink_filter : Filter
│ source_port,sink_port : PORT
│ alphabet : ℙ DATA
├───────────────────────────────────────────────────────────
│ source_port   source_filter.out_ports
│ sink_port    sink_filter.in_ports
│ source_filter.alphabets(source_port) = alphabet
│ sink_filter.alphabets(sink_port) = alphabet
└───────────────────────────────────────────────────────────
```

Software Architectures

16

Schema Pipe-State

```
┌─ Pipe_State ──────────────────────────────────────────
│ p : Pipe
│ source_data : seq DATA
│ sink_data : seq DATA
├───────────────
│ ran source_data † p.alphabet
│ ran sink_data † p.alphabet
└──────────────────────────────────────────────────────
```

Schema Pipe-Compute

```
┌─ Pipe-Compute ────────────────────────────────────────
│ Δ Pipe-State
├───────────────
│ p = p '
│ « deliver : seq DATA | #deliver > 0 ·
│     source_data = deliver ^ source_data '
│        sink_data ≐ sink_data ^ deliver
└──────────────────────────────────────────────────────
```

Schema System

. . .

```
┌─ System ─────────────────────────────────────────────
│ filters : ℙ Filter
│ pipes : ℙ Pipe
│
├───────────────────────────────────────────────────────
│   f1,f2 : filters · f1.filter_id = f2.filter_id    f1 = f2
│
│   p : pipes · p.source_filter   filters   p.sink_filter   filters
│
│   f : filters; pt: PORT | pt    f.in_ports ·
│       #{p:pipes | f = p.sink_filter    pt = p.sink_port} ≤ 1
│
│   f : filters; pt: PORT | pt    f.out_ports ·
│       #{p:pipes | f = p.source_filter    pt = p.source_port} ≤ 1
└───────────────────────────────────────────────────────
```

Software Architectures

19

Schema System-State

```
┌─ System-State ─────────────────────────────────────────
│ sys : System
│ filter_states : ℙ Filter_State
│ pipe_states : ℙ Pipe_State
│
├─────────────────────────────────────────────────────────
│ sys.filters = {fs : filters_states · fs.f}
│   fs1, fs2 : filters_states · fs1.f = fs2.f    fs1 = fs2
│
│ sys.pipes = {ps : Pipe_State ·ps.p}
│   ps1,ps2 : pipe_states · ps1.p = ps2.p     ps1 = ps2
│
│   ps : pipe_states · «fs : filter_states ·
│       ps.p.source_filter = fs.f
│           ps.source_data = fs.output_state(ps.p.source_port)
│   ps : pipe_states · «fs : filter_states ·
│       ps.p.sink_filter = fs.f
│           ps.sink_data = fs.input_state(ps.p.sink_port)
└─────────────────────────────────────────────────────────
```

Schema System-Filter-Step

```
┌─ System_Filter_Step ──────────────────────────────────
│ Δ System_State
├────────────────────────────────────────────────────────
│ sys = sys′
│ «Filter_Compute •
│     filter_states\{θ Filter_State} = filter_states ′\ {θ Filter_State }
│         θ Filter_State    filter_states
│         θ Filter_State′    filter_states ′
└────────────────────────────────────────────────────────
```

Schema System-Pipe-Step

```
┌─ System_Pipe_Step ──────────────────────────────────
│ Δ System_State
├────────────────────────────────────────────────────────
│ sys = sys′
│ «Pipe_Compute •
│     θ Pipe_State    pipe_states
│       θ Pipe_State′    pipe_states′
│       ( fil_state : filter_states;fil_state′: filter_states′I fil_state.f = fil_state ′.f
│
│         • fil_state.internal_state = fil_state ′.internal_state)
│       ( fil_state:filter_states;fil_state ′:ilter_states ;port:PORT
│           I fil_state.f = fil_state ′.f    port    fil_state.f.in_ports
│             (p.sink_filter ≠ fil_state.f    p.sink_port ≠ port)
│         • fil_state.input_state(port) = fil_state ′.input_state(port))
│       ( fil_state:filter_states;fil_state ′:filter_states ;port:PORT
│           I fil_state.f = fil_state ′.f    port    fil_state.f.out_ports
│             (p.source_filter ≠ fil_state.f    p.source_port ≠ port)
│         • fil_state.output_state(port) = fil_state ′output_state(port))
└────────────────────────────────────────────────────────
```

System Start and Final

System_Compute_Step > **System_Filter_Step**   **System_Pipe_Step**

```
┌─ SystemStart ─────────────────────────────────────
│  System_State
│  ─────────────────────────────────────────────────
│    fil_state:filter_states;port:PORT | port   fil_state.f.out_ports
│       · fil_state.internal_state.f.start
│          #(fil_state.output_state(port)) = 0
│    pipe_state : pipe_states
│       · #pipe_state.sink_data = #pipe_state.source_data = 0
└─────────────────────────────────────────────────────

┌─ SystemFinal ─────────────────────────────────────
│  System_State
│  ─────────────────────────────────────────────────
│    · («System_State'· System_Compute_Step)
└─────────────────────────────────────────────────────
```

Complete Computation

```
═══════════════════════════════════════════════════════
┌─ CompleteComputation ─────────────────────────────
│  trace : seq System_State
│  ─────────────────────────────────────────────────
│  « sys: System ·    i:dom trace · ( trace(i)).sys = sys
│  «SystemStart · θ System_State = trace(1)
│  «SystemFinal · θ System_State = trace(#trace)
│    1..(#trace-1)
│       · («Δ System_State
│          · θ System_State = trace(i) fl q System_State '= trace(i+1)
│             System_Compute_Step)
└─────────────────────────────────────────────────────
```

Experience with a Course on Architectures for Software Systems   9-12

PF Graph

PFGraph ―――――――――――――――――――
System
connect : FILTER ˄ FILTER

―――――――――――――――――――
connect = {p : pipes · (p.source_filter,p.sink_filter)}

Restrictions
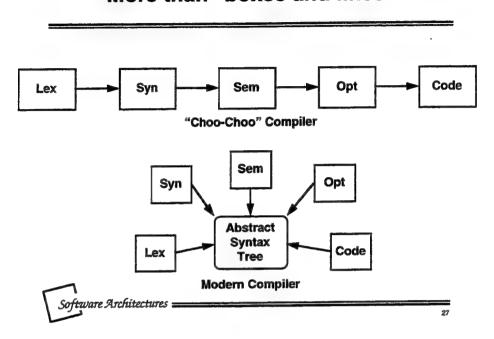
AcyclicPF―――――――――――――――――――
PFGraph

―――――――――――――――――――
  f : filters · (f,f)   connect⁺

Pipeline ―――――――――――――――――――
AcyclicPF

―――――――――――――――――――
«order : seq Filter | dom order = filters   #order = #filters
    · connect = {i:1..#filters-1 · (order(i),order(i+1))}

More than "boxes and lines"   ...

"Choo-Choo" Compiler

Modern Compiler

Software Architectures

27

Lecture 11
Communicating Process Architectures

David Garlan

Software Architectures ================================
1

Outline

- **Process Architectures**
  - > Processes, message passing
  - > Special forms: Pipe & filter, Client-server
- **Focus on Message Passing**
  - > Design issues
  - > Processing idioms
    - » Heartbeat
    - » Probe/Echo
    - » Broadcast
    - » Token Passing
    - » Replication strategies

Software Architectures ================================
2

Motivation

- **Hardware considerations lead to world of multiple processors/processes**
- **Each processor/process operates (largely) asynchronously**
- **Uses message passing for communication**
- **How can we exploit this world to get useful work done?**

Software Architectures

3

Process Architectures

- **Components: processes**
  - > Each process can be thought of as a virtual processor
  - > Operates in own address space
  - > Can communicate with the world through ports
- **Connectors: message passing over channels**
  - > Send: sends a message on a channel
  - > Receive: gets a message on a channel
- **Configurations: arbitrary graphs**

Software Architectures

4

Alternatives

- **Shared memory machines**
  - > Processes can share global variables
  - > Communication requires synchronization on these variables
  - > Algorithms are often simpler with shared memory, but may be difficult to implement efficiently and correctly
- **Remote procedure call (RPC)**
  - > Processes interact by subroutine invocation
  - > Similar to programming languages
- **Various hybrids**

Software Architectures

5

Special Cases

- **Pipe and filter**
  - > Processes read inputs and write outputs
  - > Data flows in one direction through acyclic graph
  - > We have already looked at this idiom in detail
- **Client-server systems**
  - > Client makes request to server and waits for reply
  - > Often implemented as RPC
  - > Asymmetric relationship
  - > We'll revisit this important idiom later

Software Architectures

6

Note on "Completeness"

- All of these paradigms (shared memory, client server, pipe and filter, message passing) are computationally equivalent
- Can simulate any of them using one of these paradigms
- But they have different properties
  - > performance profiles
  - > logical decomposition of problem
  - > ease of implementation
  - > efficiency for a given hardware platform

Software Architectures

7

Design Parameters for Message Passing

- Protocols of interaction
  - > synchronous/asynchronous
  - > reliable/unreliable messages/processes
  - > blocking/non-blocking
  - > buffered/non-buffered
- Topology of graph
  - > trees, rings, pipelines, acyclic/cyclic graphs
- Processing algorithm(s) of processes

Software Architectures

8

Design Considerations (1)

- **Degree of interconnectedness**
  - \> **Fully connected**
    - » algorithms are easier, since can rely on broadcast as a primitive
    - » applies to local area nets (in an ideal world)
  - \> **Partially connected**
    - » algorithms more complex, since have to worry about how to get information spread around
    - » applies more generally to all nets

Software Architectures

9

Design Considerations (2)

- **Fault model**
  - \> **Reliable nodes and channels**
    - » easier to reason about, but not always realistic
  - \> **Unreliable nodes -- but can detect when one crashes**
    - » lot's of work done in this area
  - \> **Unreliable channels**
    - » requires ack-based protocols
  - \> **"Byzantine" faults**
    - » can't detect which part of system is broken

Software Architectures

10

Design Considerations (3)

- **Simplicity of algorithm/protocol**
  - \> Symmetric
    - » algorithms run same program at each node
    - » more robust in presence of faults
  - \> Asymmetric
    - » special nodes -- e.g., root/leaves of a tree
    - » usually easier to design algorithms
    - » but harder to reason about correctness

Software Architectures

Design Considerations (4)

- **Issues of correctness**
  - \> In general, much harder than seq. reasoning
    - » but message passing actually simplifies reasoning since don't have to worry about shared state
  - \> Termination
    - » how do you know when to stop?
    - » can be difficult to know when net has reached quiescence.
  - \> Unused messages
    - » is it important to flush messages from channels on termination?
  - \> Deadlock and livelock

Software Architectures

Design Considerations (5)

- **Performance**
  - > Number of messages
  - > Size of each message
  - > Degree of asynchronicity
  - > Ability to scale up via replication
  - > Ability not to degrade as net gets bigger

Software Architectures

13

More on Message Passing Connectors

- **Channels have global names**
  - > not essential, but simplifies description of algorithms
- **Send**
  - > Usually asynchronous (non-blocking)
  - > Assume infinite buffer
- **Receive**
  - > Blocking - receiver waits until message appears
  - > Often augment with a procedure to test whether a channel is empty

Software Architectures

14

Idiom 1: Heartbeat

- **Topology: A graph with processors as nodes and communication only along edges of graph.**
- **Protocol: 2 phased rounds**
  - > Send information to all neighbors
  - > Receive information from all neighbors
- **Example:**
  - > "Life"
  - > Network topology discovery
    - » each node knows neighbors
    - » goal is to know entire structure of graph

Software Architectures

15

Network Topology (Heartbeat 1)

- **Represent network graph using adjacency matrix**
- **Each node initially has adjacency vector**
- **Shared memory is easy**
- **Distributed memory version**
  - > Send local copy of adjacency matrix
  - > Receive matrix from neighbors and update

Software Architectures

16

Network Topology (Heartbeat 2)

- **Properties of algorithm**
  - > After k rounds know topology with distance k
  - > Know full information after D rounds
    - » where D is the "diameter" of the net
  - > Wasteful of messages
    - » how can you improve it?
  - > How do you know when you are done?

Software Architectures

17

Idiom 2: Probe/Echo

- **Topology: as before**
- **Protocol**
  - > An initiator node starts by sending probe message to neighbors
  - > When receive a probe, send probe to partial set of neighbors
  - > When receive echo, send echo to set of neighbors
- **Examples**
  - > Height of tree in a tree-network
  - > Broadcast message to all nodes

Software Architectures

18

Broadcast (Probe/echo)

- Special node "I" wants to broadcast a message
- If "I" has knowledge of network topology use a (spanning) tree
- If nodes know only who are their neighbors
  - On receiving first message, send messages to all neighbors
  - Wait to receive messages from all neighbors

Software Architectures

19

Heartbeat versus Probe/Echo

- Heartbeat
  - all nodes perform same algorithm
  - implicit synchronization
  - all messages to neighbors are same
  - lots of messages
- Probe/echo
  - special node to start the computation
  - no synchronization
  - send different messages to neighbors
  - usually fewer messages

Software Architectures

20

Idiom 3: Token Passing

- **Topology: as before**
- **Protocol**
  - > Token is a special message
  - > If a process receives a token it has special permission
  - > After using the token to do something it passes it on
  - > Usually a fixed number of tokens
- **Examples**
  - > Resource contention, eg., readers/writers
  - > Termination detection in a ring

Software Architectures ======================================

21

Termination Detection in a Ring

- **Termination means**
  - > all processes are idle
  - > no messages in transit
- **Algorithm**
  - > First time process 0 becomes idle it creates a token and passes it on.
  - > If a process receives a token, it finishes up its computation and passes the token on
  - > If the process 0 has been idle throughout and gets the token again, it knows the ring is idle.

Software Architectures ======================================

22

Replication

- **Replicated computation**
  - > bag of tasks
  - > multiple servers
- **Replicated data**
  - > replicated files, for example

Software Architectures

23

Lecture 12
Communicating Sequential Processes

Robert Allen

Software Architectures ===============================

1

Overview

- **Z: a quick review**
- **Changing Point of View**
- **CSP**
- **Analyzing Processes**
- **Laws**

Software Architectures ===============================

2

The Goal of a Formal Method

- **To describe something precisely**
- **To explore its properties**
- **To communicate**

Software Architectures

3

Z (a quick review)

```
┌─ Schema ─────────────────────────────────────
│ a: ATTRIBUTE
│ b: OTHER_ATTRIBUTE
├──────────────
│ INVARIANT (a)
│ INVARIANT2(b)
│ INVARIANT3(a,b)
└──────────────────────────────────────────────
```

Z views a thing in terms of its *states*

Software Architectures

4

Changing State

Change_of_State
Δ Schema
?input
!output
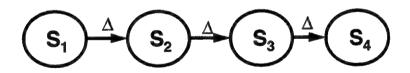
Preconditions(θ Schema,?input)
Postconditions(a,a',b,b',?input,output)

All actions are viewed as pairs of states

Software Architectures

A larger sequence of computations

$$S_1 \xrightarrow{\Delta} S_2 \xrightarrow{\Delta} S_3 \xrightarrow{\Delta} S_4$$

•States are transformed from one into another
•They are related by Δ schemas

Software Architectures

But Is that always the best way? ···

```
┌─ Customer ─────────────────────────────────────────
│ pocket_cash: ℕ
│ chocolates_eaten: ℕ
│
└─────────────────────────────────────────────────────
```

```
┌─ Machine ──────────────────────────────────────────
│ cash: ℕ
│ chocolates: ℕ
│
└─────────────────────────────────────────────────────
```

Software Architectures ══════════════════════════════

7

Some things can be described

```
┌─ Happy_Customer ───────────────────────────────────
│ Δ Customer
│────────────────────────────────────────────────────
│ chocolates_eaten' > chocolates_eaten
│ pocket_cash - pocket_cash' ≤ 75
└─────────────────────────────────────────────────────
```

```
┌─ Correct_Dispense ─────────────────────────────────
│ Δ Machine
│────────────────────────────────────────────────────
│ chocolates' = chocolates -1
│ cash' - cash ≥ 75
└─────────────────────────────────────────────────────
```
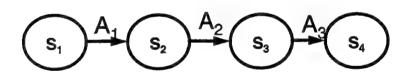
Software Architectures ══════════════════════════════

8

Others are more difficult to model ···

* **A chocolate will always be dispensed after a coin is put in?**
* **More coins may be put in, but will be returned?**
* **The machine and the customer communicate via a coin?**

Software Architectures ===================================
9

Shifting Point of View

•Actions transform states
•Describe the permitted actions

Software Architectures ===================================
10

An event represents an action

- coin!denomination
- eat_choc
- dispense_choc
- return_coin?denomination
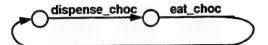
A process defines a *trace* of events

- e → P
- STOP
- RUN
- dispense_choc → eat_choc → STOP

Infinite Traces (Recursion)

- **HUNGRY =**
  dispense_choc → eat_choc → HUNGRY

Software Architectures
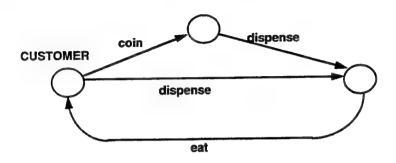
Alternatives

CUSTOMER =
    coin → EAT
    ☐ **EAT**

EAT =
    dispense?x → eat!x → CUSTOMER

Software Architectures

Alternatives

Software Architectures
15
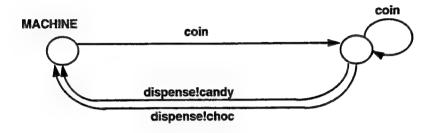
Machine

MACHINE =
   coin → (DISPENSE □ MACHINE)

DISPENSE =
   dispense!choc → MACHINE
   □ dispense!candy → MACHINE

Software Architectures
16

Machine

Software Architectures ================================= 17
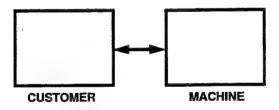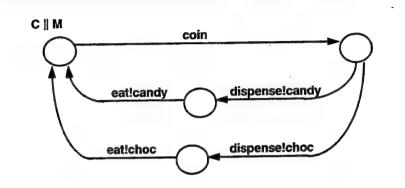
Communication

- **Customer and Machine communicate to achieve a valid transaction**

CUSTOMER || MACHINE

Software Architectures ================================= 18

CUSTOMER || MACHINE

C || M

Software Architectures ==========================

19

Customer and Machine Must Agree

CUSTOMER || MACHINE =
   coin →
         (dispense!choc → eat!choc → (CUSTOMER || MACHINE)
         □ dispense!candy → eat!candy → (CUSTOMER || MACHINE))

- **Some events are communications**
- **Why doesn't MACHINE have to agree about <u>eat</u>?**

   alphabets

Software Architectures ==========================

20

Alphabets

- α CUSTOMER =
    {coin,dispense,eat}
- α MACHINE =
    {coin,dispense}

- α CUSTOMER $\cap$ α MACHINE =
    {coin,dispense}

CUSTOMER and MACHINE only agree on the
common events

Software Architectures

21

Decision

- Shouldn't CUSTOMER decide what to eat?

PICKY_EATER =
    dispense.choc $\rightarrow$ eat.choc $\rightarrow$
    PICKY_CUST
    $\sqcap$ dispense.candy $\rightarrow$ eat.candy $\rightarrow$
    PICKY_CUST

PICKY_CUST = (coin $\rightarrow$ PICKY_EATER)
            $\square$ PICKY_EATER

Software Architectures

22

Making Claims about Processes

- **Traces**
  - $>$ traces(a $\rightarrow$ b $\rightarrow$ P) =
    - $<$ $>$, $<$ a $>$, $<$ a,b $>$, $<$ a,b,a $>$, ...
  - $>$ S(tr): $tr{\downarrow}a \leq tr{\downarrow}b$
- **Specifications**
  - $>$ P sat S(tr)
  - $>$ MACHINE sat ($tr{\downarrow}$coin $\geq$ $tr{\downarrow}$dispense)
  - $>$ CUSTOMER sat ($tr{\downarrow}$coin $\leq$ $tr{\downarrow}$dispense)
  - $>$ CUSTOMER sat *(tr*$\downarrow$eat $\geq$ $tr{\downarrow}$dispense-1)
  - $>$ C$\|$M sat $\neg$($<$coin,coin$>$ <u>in</u> tr)

Software Architectures

23

Laws

- **P sat S(tr) $\Rightarrow$ P$\|$Q sat S(tr$\upharpoonright\alpha$P)**
- **P sat S(tr) $\wedge$ Q sat S(tr) $\Rightarrow$**
  P$\square$Q sat S(tr)

- **P$\square$Q = Q$\square$P**
- **P$\square$STOP = P**
- **(P$\square$Q)$\square$R = P$\square$(Q$\square$R)**
- **(a $\rightarrow$ P) $\|$ (a $\rightarrow$ P) = a $\rightarrow$ (P$\|$Q)**
- **P $\sqcap$ (Q $\square$ R) = (P $\sqcap$ Q) $\square$ (P $\sqcap$ R)**

Software Architectures

24

Lecture 13
Models of Event Systems

David Garlan

Software Architectures

1

Outline

- Implicit invocation
- Examples
- Properties
- More examples
- Formal model

Software Architectures

2

Questions to Address

- **System Model**
  - > What is the overall organizational pattern?
- **Structure**
  - > What are the basic components and connectors?
  - > What topologies are allowed?
- **Computation**
  - > What is the underlying computational model?
  - > How is control and data transferred between components?

Software Architectures ══════════════════
3

Questions to Address (2)
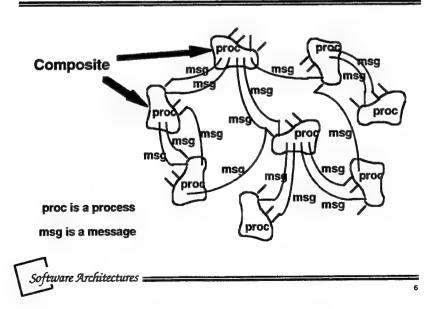
- **Properties**
  - > Why is this style useful?
  - > What kinds of properties are exposed?
- **Specializations**
  - > What kinds of variants are allowed?
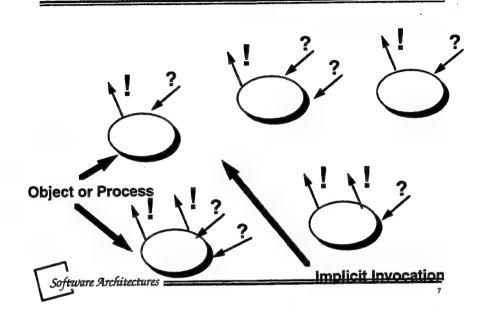
Software Architectures ══════════════════
4

Communicating Processes

- **Components: independent processes**
  - > typically implemented as separate tasks
- **Connectors:message passing**
  - > point-to-point
  - > asynchronous and synchronous
  - > RPC and other protocols can be layered on top

Software Architectures ===

5

Communicating Processes

proc is a process

msg is a message

Software Architectures ===

6

Event Systems

Object or Process

Software Architectures

Implicit Invocation

7
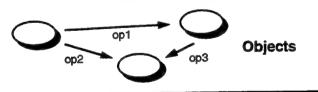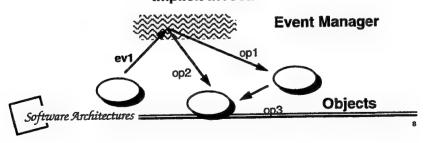
Event Systems:
Implicit versus Explicit Invocation

Explicit Invocation

op1

op2   op3   **Objects**

Implicit Invocation

Event Manager

ev1

op2   op1

Software Architectures

op3   **Objects**

8
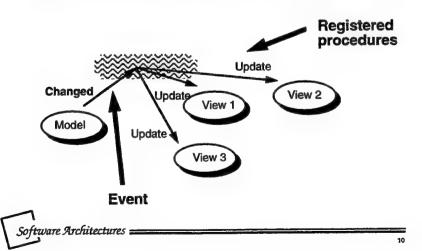
Event Systems: Model

- **Components: objects or processes**
  - > Interface defines a set of incoming procedure calls
  - > Interface also defines a set of outgoing events
- **Connections: event-procedure bindings**
  - > procedures are registered with events
  - > components communicate by announcing events at "appropriate" times
  - > when an event is announced the associated procedures are (implicitly) invoked
  - > order of invocation is non-deterministic

Software Architectures

9
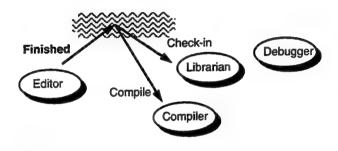
Event Systems: Example 1

Smalltalk-80 Model-View-Controller (MVC)

Software Architectures

10

Event Systems: Example 2

Field Programming Environment

Software Architectures
11
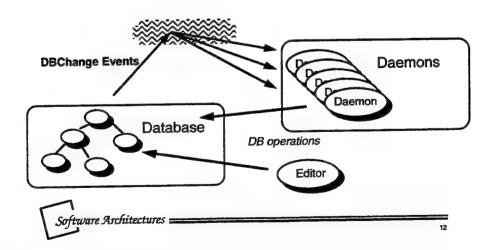
Event Systems: Example 3

Gandalf Environments

Software Architectures
12

Event Systems: Advantages

- **Problem Decomposition**
  - > Objects more independent than with explicit invocation
  - > Interaction policy can be separated from interacting objects

- **System Maintenance and Reuse**
  - > Static name dependencies not wired in so dynamic reconfiguration is easy
  - > Reuse objects simply by registering them

- **Performance**
  - > Possibility of parallel handling of events.

Software Architectures

13

Event Systems: Disadvantages

- **Problem Decomposition**
  - > No control over sequencing of invocations
  - > Function call semantics problematic
  - > Cycles may be problematic

- **System Maintenance and Reuse**
  - > Needs central management to keep track of events, registrations, and dispatch policies
  - > Event handling may interact badly with other run time mechanisms

- **Performance**
  - > Indirection may incur overhead
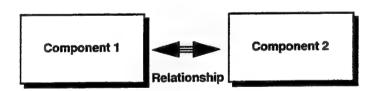
Software Architectures

14

Event Systems: Specializations

- **Tool Abstraction (e.g., Gandalf, blackboards)**
  - > Central database, events trigger daemons
- **Event-only Systems**
  - > Events are simply forwarded.
- **Dependency List (e.g., Smalltalk-80)**
  - > Each object keeps its own dependency list
- **Constraint Systems (e.g., attribute evaluation, spreadsheets, mediators)**
  - > Methods associated with events reestablish constraints.

Software Architectures ═══════════════════════

15

Application: Mediators

```
┌──────────────┐              ┌──────────────┐
│              │              │              │
│ Component 1  │ ◄══════►     │ Component 2  │
│              │              │              │
└──────────────┘ Relationship └──────────────┘
```

Example:

Component 1 = nodes of a graph

Component 2 = edges of a graph

Relationship = maintain correspondence

Software Architectures ═══════════════════════

16
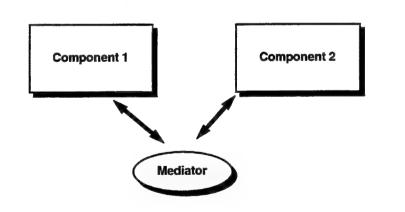
Possible Solutions

1 **Each of the components knows about the other**
> when update operation is applied, call routine in other

> result: brittle

2. **Write a third component that encompasses the two**
> new component has combined interface

> result: overly specialized

Implicit Invocation Solution
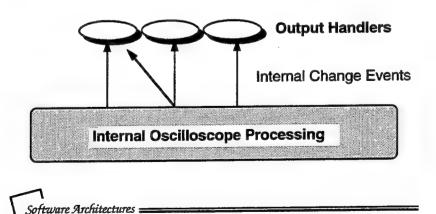
Case Study in Industrial Arch. Design

- **Recall Goals**
  - > Multiple hardware platforms for same user interface
  - > Multiple user interfaces for same platform

- **How to separate user interface from application?**

Software Architectures

19

Case Study
(continued)

Output: application announces events

Output Handlers

Internal Change Events

Internal Oscilloscope Processing

Software Architectures

20

Case Study (continued)

Input: user generates events

Input Event Dispatcher

Input Handlers

Reconfiguration Commands

Abstract Interface

Internal Oscilloscope Processing

Software Architectures ════════════════════════════
                                                    21

KWIC

Inputs: Sequence of lines

| Pipes and Filters |
| Architectures for Software Systems |

Outputs: Sequence of lines, circularly shifted
and alphabetized

| and Filters Pipes |
| Architectures for Software Systems |
| Filters Pipes and |
| for Software Systems Architectures |
| Pipes and Filters |
| Software Systems Architectures for |
| Systems Architectures for Software |

Software Architectures ════════════════════════════
                                                    22

KWIC: Solution 1 (Shared Memory)

Input → Circular shift → Alphabetize → Output

Master Control → Input, Shift, Alphabetize, Output

Input, Shift, Alphabetize, Output → Line Array, Index, Index

KWIC: Solution 2 (ADTs)

Master Control → Input, Output, Shift, Alphabetize

Advantage:
Information hiding makes
implementation changes easier
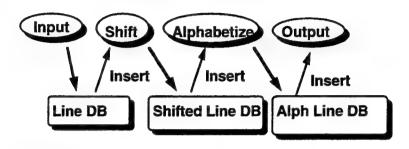
KWIC: Solution 3 (Toolies)

Interactive Version

Advantage:
Tool separation makes function
enhancements easier.

Event System Components

```
 ┌── Event-Component ──────────────
 │   name: P  NAME
 │   methods: P  METHOD
 │   events: P  EVENT
 └─────────────────────────────────
```

Event System

```
┌── EventSystem ────────────────────
│   components: $P$ Component
│   EM: Events ↔ Method
├──────────────────────
│   . . . .
└──────────────────────
```

Specialization of the Style

```
┌── Smalltalk ────────────────────
│   EventSystem
│   dependents: Component ↔ Component
├──────────────────────
│   EM = { c1,c2: components |
│           (c1,c2) ∈ dependents •
│               ((c1.name, changed), c2.name, update)) }
│
└──────────────────────
```

Experience with a Course on Architectures for Software Systems   **12-14**

Lecture 15
Event Systems:
Formal Model and Implementation

David Garlan

Software Architectures

1

Outline
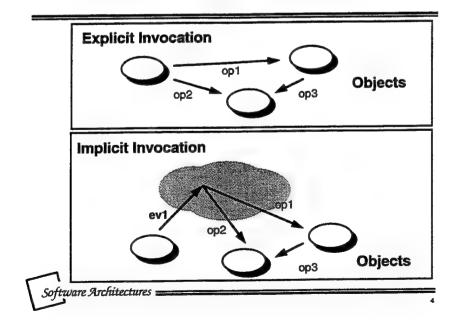
- **Review of basic properties**
- **Formal model of event systems**
- **Implementation categories**
- **Specific Implementations**
    - > The Ada-Event system
    - > Smalltalk
    - > Field
    - > Softbench
    - > Gandalf Daemons
- **Understanding the Design Space**

Software Architectures

2

Event Systems: System Model

- Components are objects or processes
- Components communicate by announcing events.
- Components register for events they are interested in and associate procedures with those events.
- When an event is announced, the registered procedures are automatically invoked.

Software Architectures

3

What is Implicit Invocation?

Explicit Invocation

Implicit Invocation

Software Architectures

4

Categories of Event System

- **Systems that support Implicit Invocation fall into three basic categories:**

  > **Programming language extensions**

    Smalltalk, Mediators, Ada Events, Toolies

  >**Tool integration frameworks**

    Field, Softbench, Forest, DecFuse, ToolTalk

  >**Special-purpose applications**

    Gandalf daemons, Active databases, APPL/A

Software Architectures ═══════════════════════
5

Event System Components

> **Event-Component** ─────────
> **name:** *P* **NAME**
> **methods:** *P* **METHOD**
> **events:** *P* **EVENT**

Software Architectures ═══════════════════════
6

Event System

> **EventSystem** ——————
> components: P Component
> EM: Events $\leftrightarrow$ Method
>
> . . . .

Specialization of the Style

> **Smalltalk** ——————
> **EventSystem**
> dependents: Component $\leftrightarrow$ Component
>
> EM = { c1,c2: components |
>   (c1,c2) $\in$ dependents •
>   ((c1.name, changed), c2.name, update)) }

Informal Analysis

- **Model allows us to predict problems**
  - \> **Burden on receiver in Smalltalk-80**
  - \> **Non-uniformity in Appl/A**
  - \> **Daemon Complexity in Gandalf**

Software Architectures

9

Formal Analysis
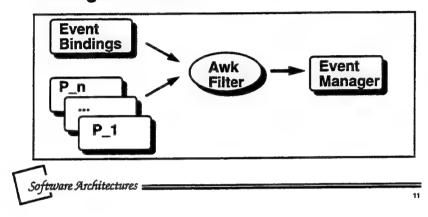
f: Field ; g: Forest $\vdash$ ... $\Rightarrow$ f.EM = g.EM

Gandalf $\vdash$ ~ Circular

Software Architectures

10

Adding Implicit Invocation to Ada

- **Annotate Ada specifications with event declarations and bindings.**
- **Use source-source filter to produce event manager in Ada .**

Software Architectures

11

Adding Implicit Invocation To Ada (1)

Event declarations

```
package Pkg_1

  --!

  declare Event_1 X: Integer; Y:
  Pkg_N.sometype;

  declare Event_2;

  --!

  procedure My_Procedure (A: Integer);

  function My_Function ....;

end P_1
```

Software Architectures

12

Adding Implicit Invocation To Ada (2)

Event-procedure bindings

```
--!
for Pkg_1
    when Event_3 => My_Procedure A
    when Event_2 => My_Procedure X
end for Pkg_1
for Pkg_2
    when Event_1 => Proc_1 Y
    when Event_2 => Proc_2
end for Pkg_2
--!
```

Software Architectures ——————

13

Adding Implicit Invocation To Ada (3)

Event announcements

```
procedure P is
  ...
  Announce_Event
     (Argument'(Event_1, X_Arg, Y,Arg));
  ...
end P
```

Software Architectures ——————

14

Event Manager (1)

Events become an enumerated type

```
package Event_Manager is
   type Event is
         (Event_1, Event_2, ...);
   type Argument (The_Event: Event) is
         record
               case The_Event is
                     when Event_1 =>
                                 Event_1_X: Integer;
                                 Event_1_Y: Pkg_N.MyType;
                     when Event_2 =>
                                 null;
   procedure Announce_Event(The_Data: Argument);
end Event_Manager;
```

Software Architectures

15

Event Manager (2)

Dispatcher is a case statement

```
with Pkg_1, Pkg_2, ...;
package body Event_Manager is
      procedure Announce_Event(The_Data: Argument) is
      begin
            case The_Data.The_Event is
                  when Event_1 =>

Pkg_2.Proc_1(The_Data.Event_1_Y);
                              ...
                  when Event_2 =>

Pkg_1.MyProcedure(The_Data.Event_1_X);
                                    Pkg_2.Proc_2

end Event_Manager;
```

Software Architectures

16

Design Issues

- **Event Declarations**
  - > Who should declare events and where?
- **Event Structure**
  - > How should events be parameterized?
- **Event Bindings**
  - > How/when should events be bound to procedures?
- **Event Announcement**
  - > How should events be announced and dispatched?
- **Concurrency**
  - > Can components operate concurrently?

Software Architectures

17

Event Declarations

- **How should events be declared?**
  - > Predefined Set of Events
  - > Static Event Declaration
  - > Dynamic Event Declaration
- **Where should events be declared?**
  - > Central Declaration of Events
  - > Distributed Declaration of Events

Software Architectures

18

Event Structure

- **How should events be parameterized?**
  - \> Simple Names
  - \> Fixed Parameter Lists
  - \> Parameters Determined by Event Type
  - \> Parameters Determined Dynamically

Software Architectures

19

Event Bindings

- **When should events be bound to procedures?**
  - \> Static Event-Procedure Binding
  - \> Dynamic Event Registration

- **How should data be communicated between an event and its implicitly-invoked procedures?**
  - \> Single Fixed Parameters (Event_Manager.Arg)
  - \> Multiple Parameters, but all passed
  - \> Selectable Parameters
  - \> Expressions over parameters

Software Architectures

20

Event Announcement

• **How should events be announced?**
  > Single Announcement Procedure
  > Multiple Announcement Procedures
  > Extend language (e.g., *announce* keyword)

Software Architectures

21

Concurrency
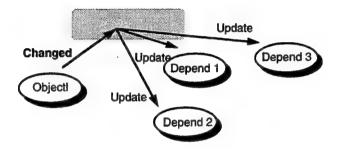
• **What is an implicitly-invocable component?**
  > Independent procedure
  > Module/object with procedure calls
  > Independent process
  > Process defined by Event_Manager

• **How are events "delivered"?**
  > Full Delivery
  > Selective Delivery
  > Pattern-based Selection
  > State-based Policy (ala Forest)

Software Architectures

22

Smalltalk

Smalltalk-80 Changed/Update Protocol
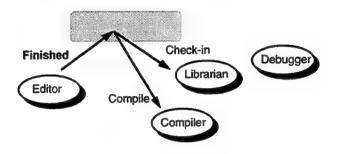
Smalltalk (continued)

- **Key Points**
  - > Commercial programing language/environment
  - > Small vocabulary of events and methods
  - > Implemented by inheritance + dependency list
  - > Synchronous dispatch
  - > Dynamic registration of dependents
  - > Primary application is user interfaces (MVC)

Field

Field Programming Environment

Software Architectures
25

Field (continued)

Key Points:

> ROTS

> Processes communicating via sockets to
central dispatcher (MSG)

> Synchronous and asynchronous

> Pattern matching as selection mechanism

> Events can be arbitrary strings

> Primary application is tool integration

Software Architectures
26

Softbench
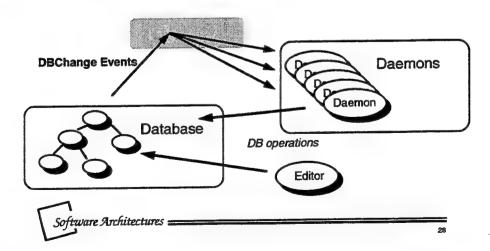
Key points:

> Commercial product (HP)

> Like Field, but

  » Events have more structure
    (tool class, context, file, ...)

  » Asynchronous only

  » Callbacks supported

> Support for "encapsulating" tools
  (UI support, message handling)

Software Architectures

27

Gandalf

Gandalf Environments

DBChange Events

Daemons

D
D
D
D
Daemon

Database

DB operations

Editor

Software Architectures

28

Gandalf (continued)

Key points:
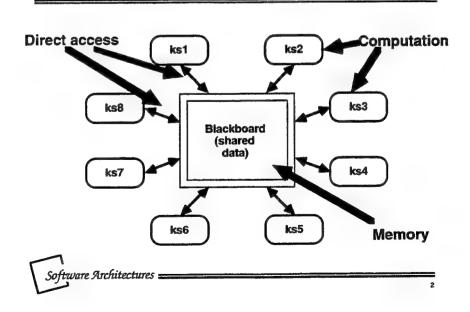
> Events triggered on operations to data

> Fixed set of events for predefined data operations

> Fixed event structure

> Extensible set of events for other operations

> Organized around transactions

> Synchronous invocation

> "Tools" are written in a special purpose language, which understands notion of events and event structure

Software Architectures

29

Lecture 16
Repositories:
Blackboard Systems

Mary Shaw

Software Architectures

1

Repository (Blackboard)

Software Architectures

2

The Blackboard Model

- **Knowledge Sources**
  - > World and domain knowledge partitioned into separate, independent computations
  - > React to changes in blackboard
- **Blackboard Data Structure**
  - > Entire state of problem solution
  - > Only means by which knowledge sources interact to yield solution
- **Control**
  - > Knowledge sources are self-activating

Software Architectures

Blackboard Architecture

- **General framework to structure and control problem-solving behavior involving multiple, diverse, and error-ful knowledge sources**
- **Independent processes achieve cooperative problem-solving**
  - > various levels of abstraction
  - > limited processing allocated to most promising actions
  - > diverse problem-solving components
  - > focus-of-control mechanism
- **Diversity ==> search among multilevel partial solutions**

Software Architectures

Blackboard Problem Characteristics

- **No direct algorithmic solution**
  - > Multiple distinct kinds of expertise
  - > Many options for what to do next
  - > Heterogeneous domain vocabulary
- **Uncertainty**
  - > Error and variability in both input & knowledge
  - > Moderate to low signal-to-noise ratio in data
  - > Uncertainty interferes with algorithmic solutions

Software Architectures

5

Blackboard Problem Characteristics

- **"Best-effort" or approximate solution often good enough**
  - > Find parts of a problem that can be solved separately
- **Large factorable solution spaces**
- **Common applications involve uncertainty**
  - > signal processing or interpretation
  - > problem-solving (e.g., planning)
  - > compiler optimization also a candidate

Software Architectures

6

Problem-Solving Models

Central question: What pieces of knowledge should be applied, and when, and how?

- **Backward reasoning:**
  - > Works from goal back to initial state
  - > Example: program verification (deterministic)

- **Forward reasoning:**
  - > Works from initial state toward goal
  - > Example: expression simplification by transformation

- **Opportunistic reasoning:**
  - > Works whichever direction seems most productive
  - > Example: trig identities

Software Architectures ================================

7

Blackboard Model, Revisited

- **Knowledge Sources**
  - > All the world & domain knowledge needed to solve problem
  - > Partitioned into separate, independent computations
  - > Respond to changes in blackboard

- **Blackboard Data Structure**
  - > Entire state of problem solution
  - > Hierarchical, non-homogeneous
  - > Modifications by knowledge sources lead to solution
  - > Only means by which knowledge sources interact

- **Control**
  - > In model, knowledge sources self-activating
  - > In framework, the magic whereby knowledge sources respond opportunistically to the state of the solution

Software Architectures ================================

8

Notes on other slides

- This lecture relies heavily on the Nii survey, so many figures from that paper are included, along with some photographs.
- At this point, extra slides are:
  - > 1. Figure with diagram of simple blackboard
  - > 2. Three photographs of koalas in eucalyptus trees (these are fairly easy to recognize)
  - > 3. Fig 4: blackboard structure for koala knowledge
  - > 4. Five more photographs in koalas in eucalyptus trees (these are much harder to recognize)

Software Architectures

9

Model -> Framework

- Add operating details to abstract model

- Purpose of framework:
  provide design guidelines for implementation in conventional computer environment

Software Architectures

10

Knowledge Sources

- **Objective:**
  - \> contribute knowledge that leads to solution
- **Representation:**
  - \> procedures, sets of rules, logic assertions
- **Action:**
  - \> modify only blackboard (or control data -- magic)
- **Responsibility:**
  - \> know when it's possible to help
- **Selection:**
  - \> loosely-coupled subtasks, or areas of specialization

Software Architectures

11

Blackboard Data Structure

- **Objective:**
  - \> hold data for use by knowledge sources
- **Representation:**
  - \> stores object from solution space, including
    - » input data, partial solutions, alternatives, final solutions, control data
    - » objects and properties define terms of discourse
    - » relationships denoted by named links
- **Organization:**
  - \> hierarchical, possibly in multiple hierarchies; need links between objects on same or different levels

Software Architectures

12

Control

- **Objective:**
  - > make knowledge sources respond opportunistically

- **Representation:**
  - > keeps various sorts of information about which knowledge sources could operate and picks a sequence that allows the solution process to proceed a step at a time

- **Remark:**
  - > the control mechanisms are thoroughly ad hoc; we will return to this problem in the next lecture with a better way to think about determining the execution order

Software Architectures

13

Notes on other slides

- **Figures from Part 2 of Nii survey:**
  - > 1. Fig 2: Hearsay task
  - > 2. Fig 3: Hearsay knowledge structure
  - > 3. Fig 4: Hearsay architecture

Software Architectures

14

Hearsay Problem-Solving Strategy

- **Bottom-up (synthetic):**
  - > interpretations synthesized from data working up abstraction hierarchy
- **Top-down (analytic):**
  - > alternatives for filling out candidate structures
- **General hypothesize-and-test:**
  - > one knowledge source generates hypothesis, another validates (prunes or assigns credibility)

Software Architectures

15

Notes on other slides

- **Figures from Part 2 of Nii survey:**
  - > 1. Fig 5: HASP task
  - > 2. Fig 6: HASP knowledge structure
  - > 3. Fig 7: HASP architecture

Software Architectures

16

HASP Problem-Solving Strategy

- **Bottom-up (synthetic):**
  > most of 40-50 knowledge sources worked bottom-up; breadth-first, pipeline style; lower-level units combined with change of vocabulary to update next level up

- **Top-down (analytic):**
  > not most numerous, but most powerful; model-driven; world view allows you to set expectations and prune out alternatives

- **General hypothesize-and-test:**
  > one knowledge source generates hypothesis, another validates (prunes or assigns credibility)

Software Architectures
17

Lecture 17
Client-Server Architectures
Databases

Jose Galmes

Software Architectures

1
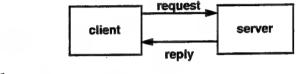
Models of Interaction (1)

- **Peer to peer**
  > Processes are independent, each executing its computation.
  > Processes occasionally communicate.
  > Either process can start the communication.
  > Processes know each other's existence.
  > Example: replicated processes.

Software Architectures

2

Models of Interaction (2)

- **Client-server**
  - > Particular case of peer to peer.
  - > Client always starts communication.
  - > Client knows server exists, but server needs not know client exists.
  - > Examples: X-Windows, NFS, Mosaic, News Readers/Servers, ...

Software Architectures

3

Why Client-Server?

- **Simplicity.**
- **Supports client computation.**
  - > and workstations are cheaper everyday.
- **In many cases, performance depends mainly on server.**
  - > Low-end workstations/PCs as clients.
- **Separation between client and server.**
  - > Easier to plug clients.
- **Redundancy at server to support fault tolerance.**

Software Architectures

4

Client-Server: disadvantages

- **Performance**
  - > Client blocks while waiting for reply.
  - > However, in many cases the client would block anyway.
- **Complexity in infrastructure**
  - > RPC mechanism.
- **Possible bottlenecks**
  - > Example: games server on WWW.

Software Architectures

5

Client-Server: types

- **Stateless**
  - > The server does not keep any state information.
  - > All the state information is in the clients.
- **State-based**
  - > The server keeps state information.
  - > Example: file server knows what clients are accessing what files.

Software Architectures

6

Client-Server: Stateless

- **Example: NFS**
- **If the server crashes, the client has all the state information.**
- **Penalty in performance.**
  - > Each interaction must carry enough information to reestablish context.

Software Architectures ================================ 7

Client-Server: State-based

- **Server has state information.**
  - > Example: knowledge of open files.
- **Server and client operate in the context of a <u>session</u>.**
- **Server more complex.**
  - > What if the server crashes?
- **Potential better performance.**
- **Some operations do not fit nicely in a stateless model (e.g. : lock())**
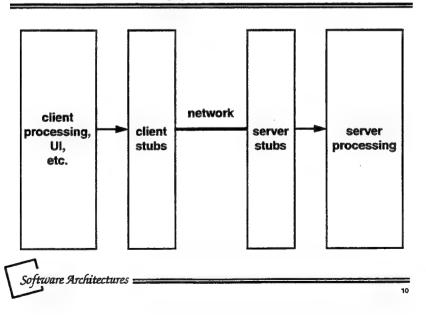
Software Architectures ================================ 8

Client-Server and RPC

- ## Client-Server vs. RPC
  - > Client-Server is the model of interaction.
  - > RPC is the most common implementation.
- ## What is RPC?
  - > RPC consists of hiding the communication protocol inside a procedure.
  - > To the client an RPC call looks like a local procedure call.

Software Architectures

9

Architecture of an RPC-based application

```
+-------------+     +---------+  network  +---------+     +-------------+
|   client    |     |         |           |         |     |             |
| processing, | --> | client  | --------- | server  | --> |   server    |
|    UI,      |     | stubs   |           | stubs   |     | processing  |
|    etc.     |     |         |           |         |     |             |
+-------------+     +---------+           +---------+     +-------------+
```

Software Architectures

10

Stub Generators

- RPC packages come with a stub generator tool.
- Given a high-level description of the protocol, the tool generates:
  - > Client code
    - » packs data, sends package to server, waits for reply and unpacks reply.
  - > Server code
    - » unpacks incoming requests, calls service routines, packs results and sends them to the client.

Software Architectures

11
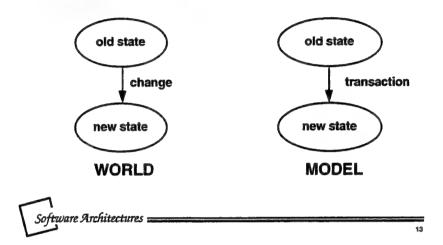
Stub generation: Example

•Separate handout.

Software Architectures

12

Transaction Processing Systems (TPS)

- **State change in model of the world mediated by transactions.**

Software Architectures

13

What is a transaction?

- **A collection of actions on the application state, obeying the ACID properties:**
  - > **Atomic:** all changes happen or none do.
  - > **Consistent:** the actions as a whole are a correct state transformation, obeying all integrity constraints.
  - > **Isolated:** transactions appear to be executed one at a time.
  - > **Durable:** once committed, changes survive failures.

Software Architectures

14

Why transactions?

- **Single failure semantics.**
- **Easier to write reliable applications.**
- **Infrastructure that can be used by many applications.**

Software Architectures

15

Basic Form of a Transaction Program

```
begin_transaction();
operation_1();
operation_2();
...
commit_transaction();
```

Software Architectures

16

TPS Architecture

* **The ACID properties suggest a need for the following functionality:**
  * **> A, C: need to undo partial computations**
    * » => Log Manager records a log of changes made by transactions, so that a consistent state can be reconstructed in case of failure.
  * **> I: need to lock/unlock objects**
    * » => Lock Manager.
  * **> D: need for permanent storage**
    * » => Resource Managers.

Software Architectures

17

TPS Architecture

Jim Gray, Andreas Reuter, "Transaction Processing, Concepts and Techniques," p. 20.

Software Architectures

18

Lecture 18
Repositories:
Information System Evolution Patterns

Mary Shaw

Software Architectures ═══════════════════════

Context

- **We previously discussed the initial form for shared information systems, batch sequential organization.**

- **We also examined two repository organization, databases and blackboards.**

Software Architectures ═══════════════════════

Batch Sequential Data Processing

Processing steps are independent programs
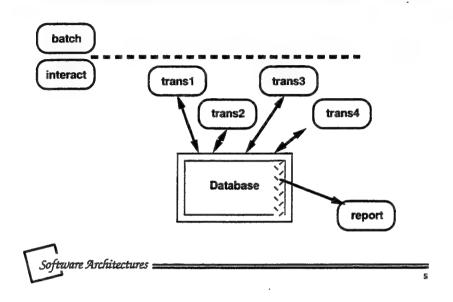
Each step runs to completion before next step starts

Software Architectures

3

Interactive Data Processing

- Laurence J. Best. Application Architecture: Modern Large-Scale Information Processing. Wiley 1990. (figures on system organization)

Software Architectures

4

Repository Architecture

Software Architectures ══════════════════════
5
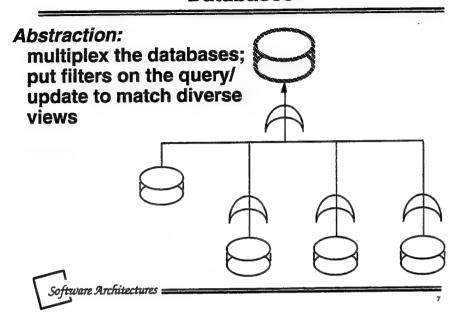
Integrating Databases

- **Won Kim and Jungyun Seo. "Classifying Schematic and Data Heterogeneity in Multidatabase Systems"** *IEEE Computer*, **December 1991, vol 24 no 12 (table 1 p.13.)**
- **Rafi Ahmed et al, "The Pegasus Heterogeneous Multidatabase system."** IEEE Computer, **December 1991, vol 24 no 12 (fig 1, p.21)**

Software Architectures ══════════════════════
6

Unified Schemas for Integrating Databases

Abstraction:
   multiplex the databases;
   put filters on the query/
   update to match diverse
   views

Computer Aided Software Engineering

- Software development
  - > Initially just translation from source to object code: compiler, library, linker, make
  - > Grew to include design record, documentation, analysis, configuration control, incrementality
  - > Integration demanded for 20 years, but not here yet
- As compared to databases:
  - > more types of data
  - > fewer instances of each type
  - > slower query rates
  - > larger, more complex, less discrete information but *not* shorter lifetime

Traditional Compiler

```
Text                                                      Code
 ──▶ [ Lex ] ──▶ [ Syn ] ──▶ [ Sem ] ──▶ [ Opt ] ──▶ [ Code ] ──▶
```

Software Architectures ══════════════════════════

Modern Canonical Compiler

```
                        [ SymTab ]

Text                                                      Code
 ──▶ [ Lex ] ──▶ [ Syn ] ──▶ [ Sem ] ──▶ [ Opt ] ──▶ [ Code ] ──▶

                         [ Tree ]
```

Software Architectures ══════════════════════════
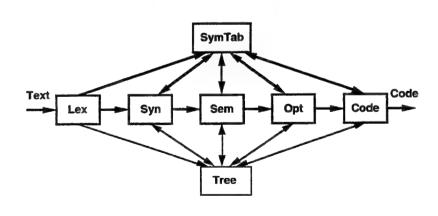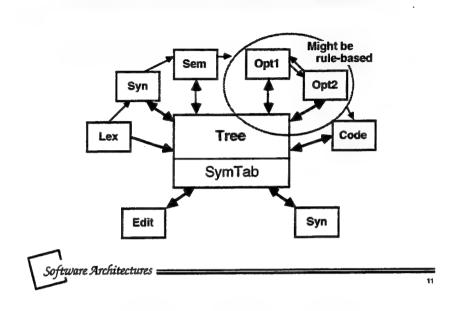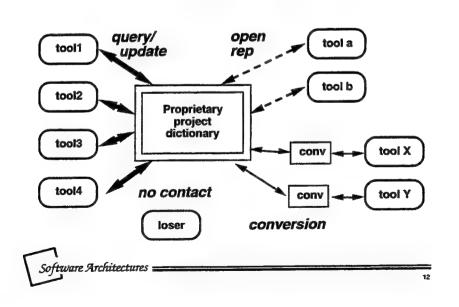
Canonical Compiler, Revisited

Software Tools with Shared Representation

Evolution of CASE Environments

- **Evolution is much like databases**
  - > Interaction: batch --> interactive
  - > Granularity: complete processing --> incremental
  - > Coverage: compilation --> full life cycle
  - > Like databases, started with batch sequential style;
  - > integration needs led to repositories with rigid control, then to open systems in layers
- **Integration still weak:**
  - > Passive conversions, rigid ordering
  - > Knowledge only of system concepts (file, date)
  - > Must learn to handle complex dependencies and selection of which tools to use, but doesn't yet

Software Architectures

13

Repositories (Review)

Control Thread Driven By	Example:
Designer (predetermined)	Compiler
Input stream	Database transaction system
State of problem solution	Blackboard

Software Architectures

14
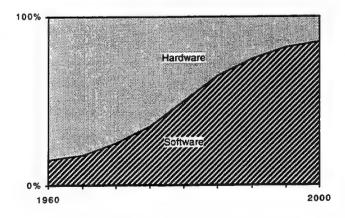
Building Big Systems from Little Ones

- **Independent components vs shared context**
  - > Consistency before the fact
  - > Support commonality via development environment

- **Open systems and proprietary architectures**
  - > Consistency through interface standards
  - > Current events and market forces

- **Distributed, dynamically open systems**
  - > Consistency after the fact

Software Architectures

15

Software Costs Dominate Computing

Software Architectures

16

The Computer Industry

- **Historical structure of computer industry**
  - > **Independent, vertically integrated competitors**
  - > **Support commonality via development environment**
  - > **Software developed for single operating environment**
- **Modern computer industry -- driven by PC revolution**
  - > **Recognizable layers**
  - > **Massive articulation between layers**
  - > **Multiple competitors in each layer**
  - > **Imperative for components to interact flexibly**
- **Object lesson: only one or two vacuum-tube companies made successful transitions to transistors and then ICs**

Software Architectures

17

Added slides

Vertical Integration in computer industry

Restructuring of computer industry

Software Architectures

18

Open Systems and Proprietary Architectures

- **New paradigm**
  For competitive success, get proprietary architectural control over a broad, fast-moving, competitive space.
- **Architectural control**
  An "architectural controller" controls one or more of the standards for assembling the entire info. package.
- **Open systems**
  Open systems are externally accessible to many vendors; critical elements are installed and deleted independently.
- **Architecture**
  The complex of standards, formats, communication protocols, and rules that define how programs and commands work and how data flows around the system.

Software Architectures
19

The Owner's Edge for an Architecture

- **Advance knowledge**
  Can start product development early.
- **Preferred directions**
  Can steer standard development to take advantage of own capabilities – or away from competitor's.
- **Competitive edge**
  Has superior understanding of how to exploit architecture.
- **... not unlimited ...**
  Unix world is now (finally!) organizing to break the Microsoft choke-hold
- **The rest of the world benefits too – architecture is not a prisoner to an international standards committee.**

Software Architectures
20

Imperatives for Architectural Competition

- **Good products are not enough**
  Must retain compatibility with growing product family
- **Implementations matter**
  Performance is a critical factor in establishing
    dominance
- **Successful architectures are proprietary, but open**
  Right degree of openness is subtle, critical decision
- **General-purpose architectures absorb special-purpose solutions**
  Successful products expand to overrun niches
- **Low-end systems swallow high-end systems**
  Hardware gets both cheaper and more powerful; users
    expand needs from low end; networks of small
    systems are increasingly powerful and flexible.

Software Architectures

21

Adobe

- ## Roots in Xerox PARC
  > Interpress: exchange format for printer
    flexibility

- ## Competitive basis: Postscript, fonts
  > Postscript open (worked) but fonts closed
    (failed)

- ## Continuing product development
  > 15,000 typefaces; Type Manager; Illustrator;
    Photoshop; Premiere; PixelBurst coprocessor;
    UNIX & PC support

Software Architectures

22

Adobe (2)

- **Printing industry standard**
  - > Originally low-end printers; now imagesetters for publishing industry; ISO page-description standard

- **Next generation products under development**
  - > Postscript for Fax: remote printer as well as fax machine
  - > Acrobat: storage, compression, transmission for true document interchange
  - > Cooperative development program with OEMs

Software Architectures

23

System Incompatibility Problems

- **Technology fruit salad**
  - > Individually attractive products create competing guilds

- **Turnkey virus**
  - > Bundled "business solutions" proliferate gratuitous diversity

- **Standard vendors vs platform standards**
  - > 4 architectures from 1 vendor vs 1 architecture from 4 vendors

Software Architectures

24

System Incompatibility Problems (2)

- **Technology balkanization**
  - > Internal politics creates opportunities for incompatibility
- **Trojan horse consulting**
  - > Beware "free" consulting services from your vendor
- **Outsourcing and entropy**
  - > Outside contractor's interests may not match yours

Software Architectures

25

The "Open Architecture" Edge

- **Monopolies are no longer practical**
  - > Need to evolve
  - > Need to accommodate multiple technologies
  - > Better to have large share of a big shared market segment than fragile private slice of whole market
- **The edge is in lead time, not private access**

Software Architectures

26
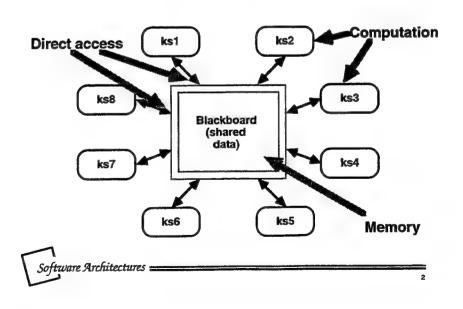
Lecture 19
Mixed Use of Idioms in
Software Architectures

Mary Shaw

Software Architectures

Repository Pattern (Blackboard)

Software Architectures

Interpreter Pattern

Memory

Inputs → Data (program state)

Computation state mach

Program Being Interpreted

Outputs ← Simulated Interpretation Engine

Selected instruction
Selected data

Internal Interpreter State

Data access
Fetch/store

Software Architectures

3

Level 5
Corp Comp Mgmt
Computers SE

Interface to Host Computers
Level 4
Plant Management

PROVOXplus Application Software
Level 3
Process Management

ENVOX Configuration Software
Level 2
Process Supervision

PROVUE Consoles
Level 1

PROVOXplus Controllers
Process Measurement and Control

Software Architectures

4

Experience with a Course on Architectures for Software Systems   17-2

TAG
TAG
TAG
TAG

DATA	ACTION	SERVICES
OPERATING TUNING CONFIGUR-ATION	ALGORITHM NAME(S)	COMMUNI-CATION TRACE ALARMS ETC
	TEMPLATE	

Software Architectures

5

Simple Rule-Based System

Knowledge base

Inputs → Working memory — Triggering data

Rule memory | Fact memory

Data | Updates

Rules and facts

Rule interpreter ← Selected rule — Rule and data element selection

Outputs ← | Selected data

Software Architectures

03191CS

6

Experience with a Course on Architectures for Software Systems   17-3

Software Architectures

7

Software Architectures

8

Simple Blackboard

Blackboard

Knowledge Source

Knowledge Source

Knowledge Source

(—— Data flow)

Software Architectures

03191CS

9

Figure 7: Hearsay-II Architecture
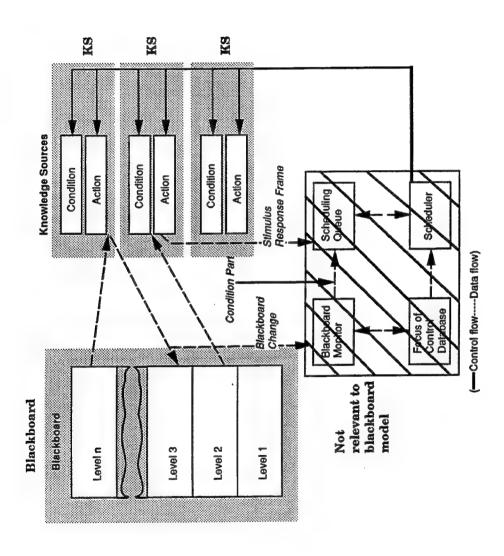
Figure 8: Blackboard View of HEARSAY-II
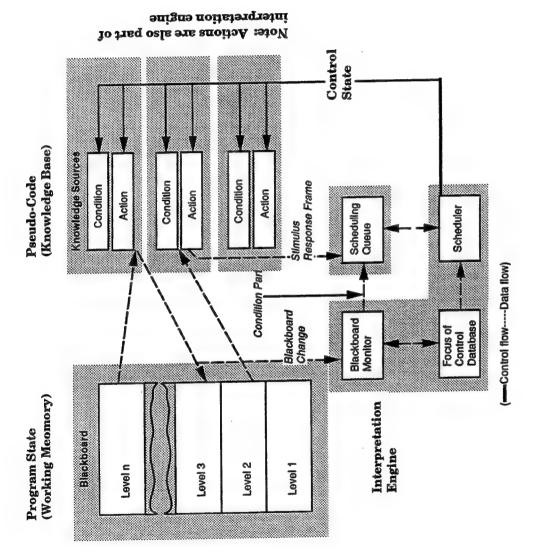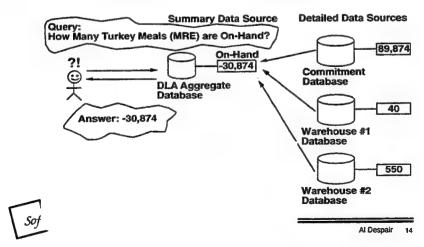
Figure 9: Interpreter View of HEARSAY-II

Your **System**
is
My **Component**

Software Architectures ════════════════════════════

Example: "Meals Ready to Eat" (MRE)
UNDERSTANDING THE MEANING OF DATA

Sof

Al Despair   14

Future Information Systems

- **High-speed networking provides ever-faster access to ever-more data**
- **Problems for single databases**
  - > Sheer volume of available data
  - > Lack of abstraction
  - > Need to understand representation of data
- **Problems for combining multiple databases**
  - > mismatch of information representation and structure
- **Knowledge vs data**
  - > *Data:* specific instances and events; gathered clerically or mechan.; correctness can be checked
  - > *Knowledge:* abstract classes, each covering many instances; requires expertise

Software Architectures ===
15

Mediation

- **Transformation and subsetting of databases using view definitions and object templates**
  - > Reorganize base data into new configurations
- **Methods to gather an appropriate amount of data**
  - > Deal with recursively linked data, temporal granularity, detail/generalization shifts
- **Methods to access and merge data from multiple databases**
  - > Compensate for mismatch of database structure, representation

Software Architectures ===
16

Mediation

* **Abstraction and generalization over underlying data**
  > Raise level of detail: statistical summarization, searching

* **Extraction of information from structured text**

* **Maintain derived data**
  > Maintain integrity as originating databases change

Software Architectures

17

You Can't Call
Your *DBMS*
As a Subroutine

Software Architectures

18

Mediator Architecture
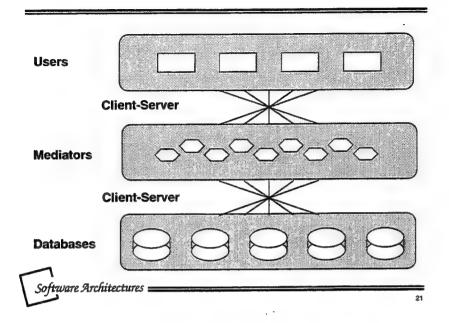
- **Architectural drivers**
  - > Distributed databases with repres. mismatches
  - > Maintenance of derived abstractions; data fusion
  - > Highly dynamic collection of available components: need flexibility

- **Layered architecture**
  - > Separate user applications from data resources with mediator layer
  - > Dynamic interfaces between layers are most critical
  - > Layers segmented internally
  - > Event triggering for dynamic response

Software Architectures

19

Mediator Architecture
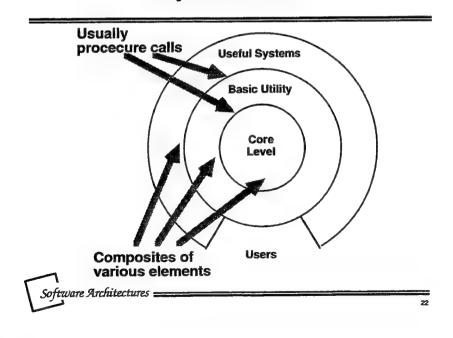
- **Architectural considerations for mediators**
  - > Most user tasks will use multiple mediators
  - > Each mediator will use one or a few databases
  - > Mediators must be inspectable for validation or selection
  - > Mediators must be dynamic, able to create many views
  - > Mediator definitions must cascade: metamediators

Software Architectures

20

Multi-Databases

Users

Client-Server

Mediators

Client-Server

Databases

Software Architectures

21

Layered Pattern

Usually procecure calls

Useful Systems

Basic Utility

Core Level

Composites of various elements

Users

Software Architectures

22

Example: "Meals Ready to Eat" (MRE)
UNDERSTANDING THE MEANING OF DATA

Summary Data Source   Detailed Data Sources

Query:
How Many Turkey Meals (MRE) are On-Hand?

?!

☺

On-Hand
-30,874

DLA Aggregate
Database

89,874

Commitment
Database

40

Warehouse #1
Database

550

Warehouse #2
Database

Answer: -30,874

Some Issues:
1. What does the answer mean?
   On-hand quantity =
      Count-on-shelves - committed (derived data)
2. How to construct the answer in compliance
   with the user's desired meaning?

Software Architectures

Al Despair   23

Environment Integration

- ## Component independence
  - > Components should be usable in different configurations
  - > Source code should not depend on other sources
  - > Relationships should not be exclusive

- ## Sources of trouble
  - > Composition via encapsulation hides components
  - > Relationships are encoded in the interacting components

Software Architectures

24

Environment Integration (2)

- **Event-method relations**
  - > Decouple caller & responder from call/ response association
- **"Mediator"**
  - > First-class component that maintains relationships
  - > Maintain state; call other components with side effects
  - > Export abstract interfaces, announce events

Software Architectures
25

Environment Integration (3)

- **Sullivan and Notkin event solution**
  - > Achieve integration by enforcing a single interaction discipline
  - > This is a common approach; we've seen several such
- **Dealing with foreign components**
  - > You often want to reuse components that don't follow the rules
  - > Here, you create a mediator for each such component that uses functionality of the available interface and exports an interface in the proper form -- "wrappers"

Software Architectures
26

Environment Integration (4)

- **Timing and pacing**
  - > Mediators as separate components add a layer of calls
  - > Since mediators can retain state, they can hold data in buffers and delay computations ("lazy evaluation")

Evolution of Database Architectures
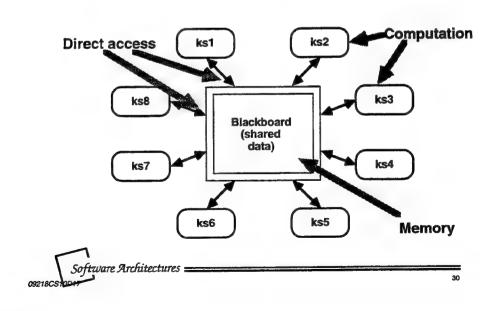
- **Batch processing**
  - > Standalone programs; results were passed from one to another on magtape; *batch sequential model*

- **Interactive processing**
  - > concurrent operation and faster updates preclude batching, so updates are out of synch with reports. *Repository model with external control*

- **Information became distributed among many different DBs**

- **Unified schemas**
  - > create one *virtual database* by defining (passive) consistent conversion mappings to multiple DBs

Evolution of Database Arch. (2)

- **Multi-database**
  - \> DBs have many users
  - \> passive mappings don't suffice
  - \> use active agents
  - \> *Layered hierarchy*

- **Progress is limited by volume, complexity of mappings and need to handle data discrepancies**

Software Architectures

29

Repository Pattern (Blackboard)

Direct access   ks1   ks2   Computation

ks8   Blackboard (shared data)   ks3

ks7   ks4

ks6   ks5   Memory

Software Architectures

30

Repository Pattern (Blackboard)

- **General framework to structure and control problem-solving behavior involving multiple, diverse, and error-ful knowledge sources**
- **Independent processes achieve cooperative problem-solving**
  - > various levels of abstraction
  - > allocation of limited processing to most promising actions
  - > diverse problem-solving components
  - > focus-of-control mechanism

Software Architectures

10048CS98173

31
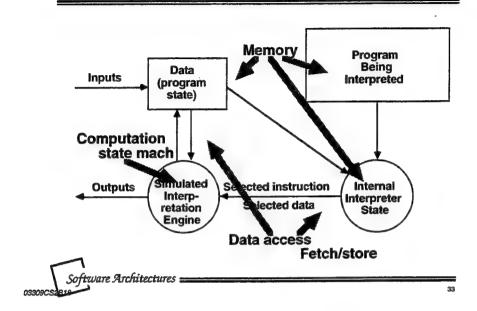
Repository Pattern (Blackboard) (2)

- **Diversity ==> searching among multilevel partial solutions**
- **For blackboard, control is data-driven (external); for other repositories, control is predetermined (internal).**

Software Architectures

32

Interpreter Pattern
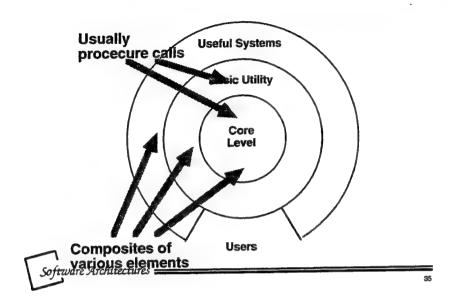
Interpreter Pattern

- **Execution engine simulated in software**
- **Data:**
  - > representation of program being interpreted
  - > data (program state) of program being interpreted
  - > internal state of interpreter
- **Control resides in "execution cycle" of interpreter**
  - > but simulated control flow in interpreted program resides in internal interpreter state
- **Syntax-driven design**

Layered Pattern

Usually procecure calls

Useful Systems

asic Utility

Core Level

Composites of various elements

Users

Software Architectures

35

Layered Pattern

- **Each layer provides certain facilities**
  - > hides part of lower layer
  - > well-defined interface
- **Serves various functions**
  - > kernels: provide core capability, often as set of procedures
  - > shells, virtual machines: support for portability
  - > client/server hierarchy: new (more abstract) service at each layer

Software Architectures

36

Building Design

- **Construction industry**
  - > Well-established decomposition of responsibilities
  - > Geographically dispersed solutions to subproblems
  - > Different collection of organizations each time
  - > Tasks interact, and coordination is its own specialty

Software Architectures

37

Building Design (2)

- **Computing evolved bottom-up**
  - > Next big step is entire facility development process
  - > Algorithmic systems for designs in individual subindustries
- **Third of three examples: stages before standalone interactive systems similar to other examples --> pick up from early integration efforts**

Software Architectures

38

Integrated Building Design Systems

- **Selection and composition of individual tool results requires judgment, experience, and rules of thumb**
  - > Not algorithmic
  - > Requires planning
- **Early efforts: support-supervisory systems**
  - > Add data management, information flow control to tools
- **Goal is integration of data, design decision, knowledge**
  - > Closely-coupled Master Builder, or
  - > Design environment with cooperating tools

Software Architectures
39

Problem-Solving for Design Control

- **Many attempts in '80s**
- ***Data:*** **mostly repositories: shared common representation with conversions to private representations of the tools**
- ***Communication:*** **mostly shared data, some messaging**
- ***Tools:*** **split between closed (tools specifically built for this system) and open (external tools can be integrated)**

Software Architectures
40
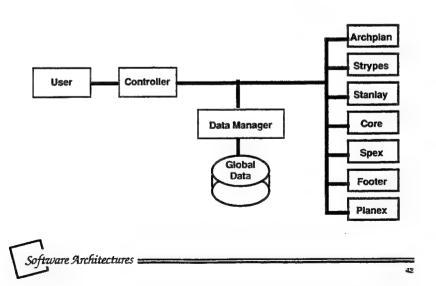
Problem-Solving for Design Control (2)
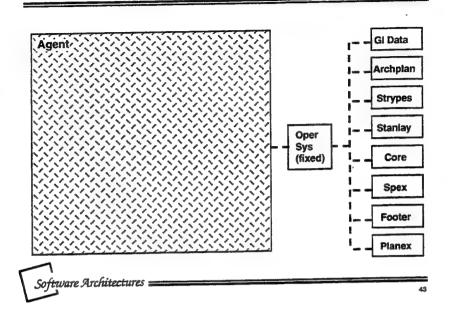
- *Control:* mostly single-level hierarchy; tools at bottom; coordination at top
- *Planning:* mostly fixed partitioning of kind and processing order; scripts sometimes permit limited flexibility

Software Architectures
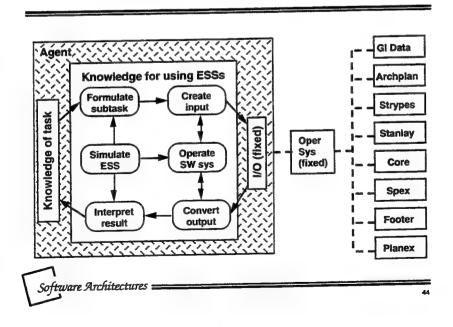
Integrated Building Design Environment

Software Architectures

Intelligent Control of IBDE

Software Architectures

43

Intelligent Control of IBDE

Software Architectures

44

Experience with a Course on Architectures for Software Systems

Lecture 20
Innovations in Module
Interconnection Languages

Robert DeLine

Software Architectures

1

State of the Course

- **Overview [3]**
- **Architectural Idioms**
  - > Procedure call [3]
  - > Data flow [4]
  - > Processes [2]
  - > Events [3]
  - > Repositories [3]
  - > Interpreters and heterogeneous systems [1]
- **Describing architectural configurations [4]**
- **Specific architectures [3]**
- **Design guidance [1]**

Software Architectures

2

Describing architectural configurations

- *Classical module interconnection languages [Lecture 3]*
- **Newer module interconnection languages**
- **Interface matching**
- **Connection languages**
- **Connection formalisms**

Software Architectures

3

Newer MILs: Overview

- **A quick review of MILs**
- **Formalizing and expanding MILs**
  DeWayne Perry
- **"Coordination" languages**
  Victor Mak
  David Galernter and Nicholas Carriero
- **How are coordination languages like MILs?**

Software Architectures

4

A Quick Review of MILs

- **A system is composed of modules that import and export resources**
  - > Resources: functions, variables, constants, ...
  - > Composition: systems can be sub-systems
- **Tools ensure system integrity**
  - > Imports match exports?
  - > Type checking
  - > Access control

Software Architectures

5

Software Interconnection Models

- **Each model can be described as a pair:**
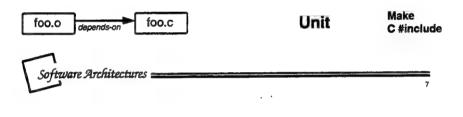
  $\langle$ {objects}, {relations} $\rangle$

  Can be visualized as a graph with labeled arcs
- **Perry presents three models:**
  - > Unit
  - > Syntactic
  - > Semantic
  - > Each model is richer than the previous by allowing more objects and relations
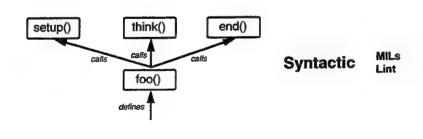
Software Architectures

6

Unit I.M.

foo.o →[depends-on] foo.c   **Unit**   **Make**
                                                      C #include

Software Architectures ═══════════════════════
                                                         7

+ Syntactic

```
setup()    think()    end()
     ↖calls  ↑calls  ↗calls         Syntactic    MILs
          foo()                                   Lint
      defines ↑
```

Software Architectures ═══════════════════════
                                                         8

+ Semantic

satisfies   satisfies

| open(f) | closed(f) | open(f) | closed(f) |

Semantic   Inscape

post   *oblig*   *pre*   *post*

Software Architectures

9
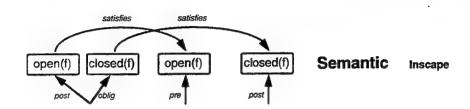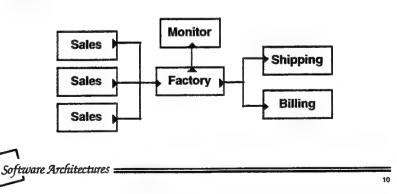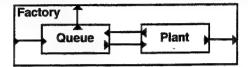
Composing Distributed Systems

- **Today's distributed systems are brittle**
  - > One subsystem directly references another
  - > Very little abstraction
- **Mak's solution: Connection**

Sales
Sales
Sales
Monitor
Factory
Shipping
Billing

Software Architectures

10
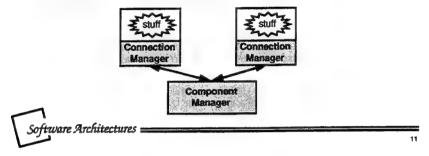
Connection: A Few Details

- **Composite Components: Scalability**
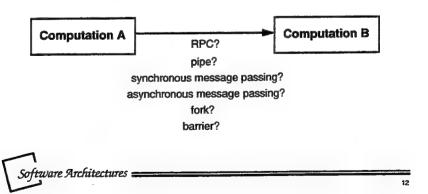
- **Component Manager ⇒ Name Server**

Software Architectures

11

Gluing Computations Together

- **Coordinating computations**
  - > Data and control exchange
  - > Diversity


```
Computation A          Computation B
              RPC?
              pipe?
    synchronous message passing?
    asynchronous message passing?
              fork?
              barrier?
```

Software Architectures

12

Gluing Computations Together

> **Complexity**

Software Architectures ————————————————
13
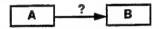
Galernter et al's Solution

- **Connections deserve to be first class citizens**
- **Galernter et al's thesis**
  - > **Connections should be expressed in a "coordination" language**
  - > **Linda is a good choice**

Software Architectures ————————————————
14

But How Are These MILs?

- **Both describe SW component and their connections**
  - \> MILs
    - » Components: Modules, Functions, Variables, ...
    - » Connections: Is-Compose-Of, Calls, Exports, ...
  - \> Linda and Connection
    - » Components: Computations, Processes
    - » Connections: Data and Control communication

$$ \boxed{A} \xrightarrow{\ ?\ } \boxed{B} $$
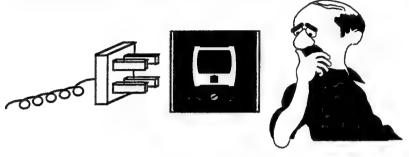
- **So is there something more general?**

Stay tuned...

Software Architectures

15

Lecture 21
Component Composition and Adaptation

Jose "Pepe" Galmes

Software Architectures

1
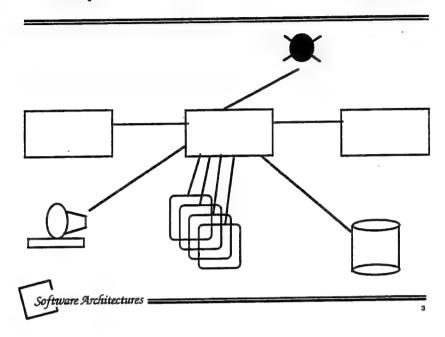
Overview
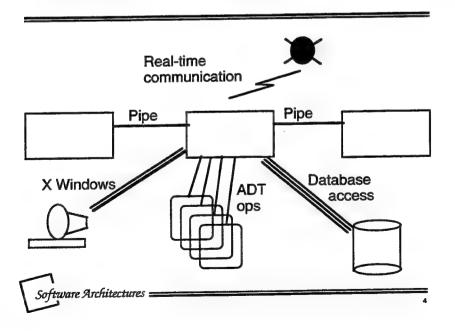
- **Combining systems:**
  - > Component interactions
  - > Current MILs
  - > Design Languages
- **Component Reuse**
- **Interface Matching**
  - > Nimble
  - > Bart
  - > Negotiated Interfaces

Software Architectures

2

Component Interactions Aren't All Alike

Software Architectures — 3

Component Interactions Aren't All Alike

Real-time
communication

Pipe   Pipe

X Windows   ADT   Database
ops   access

Software Architectures — 4

Current Module Interconnection Languages

- **Assume module structure like Ada, Modula, Cedar**
- **Support visibility control for names**
  - > *provides/requires*
  - > various granularities (entire modules to subfields)
  - > reaction to block structure
- **Mostly support single kind of interconnection**
  - > usually procedure call
  - > others include data flow (unix shell)

Software Architectures

5

Current Module Interconnection Languages (2)

- **Mostly deal only with access rights and type checking**
- **Often support a single language**
- **Have no way to take advantage of special properties**
  - > (*e.g.*, an abstract data type is specified as algebra)

Software Architectures

6

Creating Systems

- **Subsystems may be composite or primitive**
    - \> Composites are like systems
        - » Large designs require structure
        - » They can be composed of subsystems
        - » Different organizations can be used at different times
    - \> Primitives at the architecture level are programs   at a lower level
        - » Roughly at scale of a module
        - » Built in conventional programming languages

Software Architectures

Structural/Functional Elements

- **Computation:** *simple in/out relations, no retained state*
    - \> math. funct., filters, transforms, transducers
- **Memory:** *(shared) collection of persistent structured data*
    - \> data base, symbol table, file system, directory, array, hypertext
- **Manager:** *state and closely related operations*
    - \> abstract data type, resource manager, many servers

Software Architectures

Structural/Functional Elements (2)

- Controller: *governs time sequences of others' events*
  - > scheduler, synchronizer
- Link: *Transmits information between entities*
  - > communication link, remote procedure call, user interface
- Command system: *discrete, repeated, usually local, syntax-intensive manipulation of an entity*
  - > editors, operating systems, menu systems

Software Architectures

9

Combining Subsystems

- Mechanisms for connecting subsystems:
  - > Procedure call
  - > Data streams
  - > Instantiation
  - > Data sharing (direct access)
  - > Message passing
  - > Implicit triggering

Software Architectures

10

Combining Subsystems (2)

- **Interface protocols:**
  - > Calling sequences
  - > Addressing assumptions
  - > Formal protocols
  - > Shared representations

Software Architectures

Critical Elements of Design Language

- **Components**
  - > Module-level elements, not necessarily compilation units
  - > Function shared by many applications

- **Operators**
  - > For combining design elements

- **Abstraction**
  - > Ability to give names to elements for further use

Software Architectures

Critical Elements of Design Language (2)

- ## Closure
  - > Named element can be used like primitives

- ## Specification
  - > More properties than computational functionality
  - > Specs of composites derivable from specs of elements

Software Architectures

13

Name Matching and Embedded Connections

```
Main
use Foo
...f(y)...
...x...
```
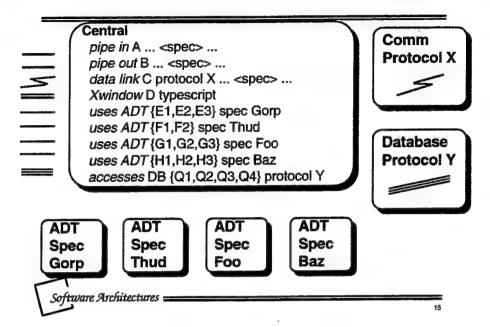
```
Foo
var x
function f(z)
some specs
```

```
Baz
var y
function g(w)
specs=foo's
```

```
OneStruc
components
  Main, Foo
procedure
  main.f calls Foo.f
var
  main.x uses Foo.x
```

```
OtherStruc
components
  Main, Baz
procedure
  Main.f calls Baz.g
var
  Main.x uses Baz.y
```

Software Architectures

14

Make Component Interactions First Class

Central
- *pipe in* A ... <spec> ...
- *pipe out* B ... <spec> ...
- *data link* C protocol X ... <spec> ...
- *Xwindow* D typescript
- *uses ADT* {E1,E2,E3} spec Gorp
- *uses ADT* {F1,F2} spec Thud
- *uses ADT* {G1,G2,G3} spec Foo
- *uses ADT* {H1,H2,H3} spec Baz
- *accesses* DB {Q1,Q2,Q3,Q4} protocol Y

Comm Protocol X

Database Protocol Y

ADT Spec Gorp **ADT Spec Thud** **ADT Spec Foo** **ADT Spec Baz**

Software Architectures

15

Component Reuse

- **Assumption behind reuse:**
  - > the components will be (re)used often enough to justify the expense of packaging, distributing, finding, and using it.

- **Most components have quite specific interfaces, including details such as**
  - > parameter types and orders,
  - > explicit naming of type substructure in parameters,
  - > style of announcing exceptions, and
  - > general form of interaction.

Software Architectures

16
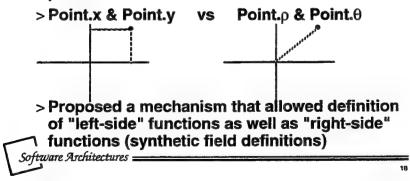
Theory vs Practical Use

- **The component you're reusing isn't always packaged in exactly the right form. Problems may involve:**
    - > Parameter order, parameter names
    - > Protocol, calling sequence
    - > Representation: right data, but in wrong order
    - > Representation: right semantics, but wrong representation
    - > Nature of interaction

Software Architectures

17

"Uniform Referent"

- **[Geshcke and Mitchell, 1975]**
    - > User of data shouldn't know its representation
    - > Accessors often reveal this: A[i] vs A.i
    - > Further, more than one representation is possible:
    - > **Point.x & Point.y** vs **Point.ρ & Point.θ**

    - > Proposed a mechanism that allowed definition of "left-side" functions as well as "right-side" functions (synthetic field definitions)

Software Architectures

18
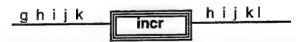
New York Public Library, revisited

- **Suppose you want subject, author, and LC call number**
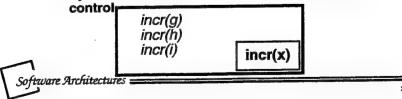
CDB1:	1, 4	item.subject
	1, 3	item.author-name
	2, 2-4	*concat*(lc-num.c-letter, lc-num.f-digit, lc-num.s-digit)
CDB2:	2, 2	item-subject.subject
	1, 3	items.a-name
	1, 5-7	*concat*(items.c-letter, items.f-digit, items.s-digit)
CDB3:	1, 5	books.subject
	1, 3	books.name
	1, 2	books.lc-num
CDB4:	- -	<nil>
	1, 4	item.a-name
	1, 2	item.lc-number

Software Architectures

Interface vs Base Functionality

> Actual utility of a component depends on the way it's packaged as well as what it computes.

> Example:

» UNIX supplies same functionality packaged both as filters and system calls

» *Filter:* incremental processing on a stream

g h i j k | incr | h i j k l

» *System call:* one call per item, single thread of control

```
incr(g)
incr(h)
incr(i)          incr(x)
```

Software Architectures

What to do when interfaces do not match?

- **Rewrite one of the modules**
- **Adapt the interfaces**

Software Architectures

21

Why Interface Adaptation?

- **Reduce development costs**
  - > Interface adaptation can be done automatically.
- **Only object code is available.**
  - > No chance to modify the components.
- **Simpler components**
  - > No need for extra interfacing code.
- **Less error-prone**
  - > No need to change existing code.

Software Architectures

22

Why Interface Adaptation? (2)

- **Configuration Management easier.**
  - > No need to keep slightly different versions of the same component.
  - > No need for revalidation of components.
- **Concentrate on the "real" application.**

Software Architectures

23

Interface Adaptation: disadvantages

- **Performance**
  - > Extra code to do the conversions.
  - > Conversions can be expensive
    - » Example: slicing an array
    - » Example: restructuring large data structure.

Software Architectures
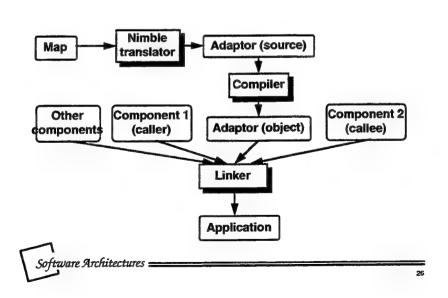
24

Purtillo and Atlee: Nimble

- Declarative language.
- **Maps** define transformations of actual parameters to match formal parameters at runtime.
- Generates implementations of maps **(adaptors)**.
- Adaptors are integrated into the application.

Software Architectures

25

Creating applications with Nimble

Software Architectures

26

Nimble - additional features

- **Interface can be automatically extracted from source code.**
- **Nimble translator checks that range of map matches formal pattern.**

Software Architectures
27

Nimble (cont.)

- **Primary expectations:**
  - > **Re-order parameters.**
  - > **Select and rearrange fields of records.**
  - > **Slice arrays.**
  - > **Simple type coercion.**
  - > **Add constants or simple expressions over actual parameters.**

Software Architectures
28

Nimble - completeness

- *Algebraic notation is complete, but*
- *Trapdoor:* EVAL, general evaluation function
  - > EVAL(user-provided-transformer, param-list)
  - > Used when algebraic notation is not practical
    - » example: slice array
  - > Or for efficiency reasons
  - > "Recursive reuse" of common transformations.

Software Architectures

29

Beach: Bart Software Bus

- A software bus is a mechanism for connecting software components.
- Analogous to a hardware bus.
  - > Allows communication among components that follow a standardized protocol.
- Ability to "plug in" new components easily.
  - > Component independence.
- Usually implemented as multi-cast or broadcast communication.

Software Architectures

30

Bart software bus

- Intuition: object-oriented components with DB relations for data interchange.
- Bart is organized in 3 levels of abstraction:

Mappings between relations (active)
Object reps cast as relations
Multicast message passing

Software Architectures

31

Bart: Message Transport

- Multi-cast approach
- Each component indicates the messages in which it is interested.
- When a messages is sent it is delivered to all interested parties.
- Callbacks used to service messages.
  - > When a component registers interest in a message, it provides a callback function.
  - > The callback function is invoked whenever the message is received.

Software Architectures

32
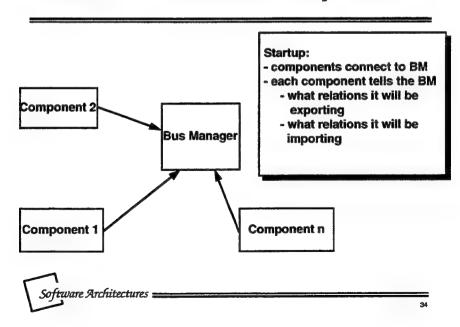
Beach: Bart software bus (2)
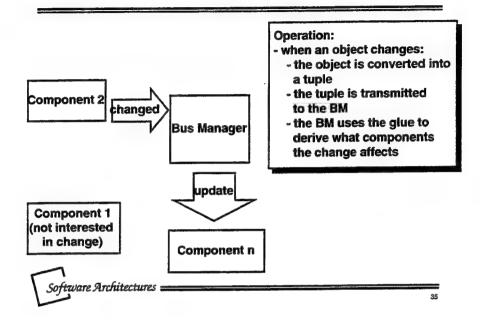
- **Each object has publisher and multiple subscribers**
  - > subscribers have shadow copies that are automatically updated.
- **General database operations support mappings:**
  - > renaming,
  - > selection,
  - > filtering,
  - > summarization,
  - > collection

Software Architectures

33

Bart: but how does it really work?

Component 2

Bus Manager

Component 1

Component n

Startup:
- components connect to BM
- each component tells the BM
  - what relations it will be exporting
  - what relations it will be importing

Software Architectures

34

Bart: but how does it really work? (2)

Component 2 changed → **Bus Manager**

Operation:
- when an object changes:
  - the object is converted into a tuple
  - the tuple is transmitted to the BM
  - the BM uses the glue to derive what components the change affects

Bus Manager → update → **Component n**

Component 1 (not interested in change)

Software Architectures

35

Novak et al: Negotiated interfaces

- **2 methods for semi-automatic interface conversion:**
  - **1. LINK: generation of a conversion program**
    - » Examine subroutine's expectations and fields provided by caller
    - » Propose possibilities for matches to user; generate code
    - » Tuned for case where data is just rearranged
    - » Works for input parameters only.

Software Architectures

36

Experience with a Course on Architectures for Software Systems   **19-18**

Novak et al: Negotiated interfaces (2)

2. instead of a real subroutine, the callee is initially a generic algorithm with abstract data as arguments;

    » the system generates a specialized version of the algorithm

    » the generated subroutine directly operates on the application data.

Software Architectures

37

Novak et al: Negotiated interfaces (2)

- In both cases, specification is produced via a menu-based negotiation with the user.

- Based on GLISP, which has mechanisms for uniform reference.

Software Architectures

38

Analysis

- **All three tackle the problem of mild data mismatch across interfaces.**
- **Novak et al and Purtilo&Atlee support a procedure call model.**
- **Novak et al believe that added components are inefficient**
- **Purtilo&Atlee want to work without source code**

Software Architectures

39

Analysis (2)

- **Novak et al aspire to richer mappings**
- **Purtilo&Atlee believe that simple mappings handle most cases and provide trapdoor.**
- **Beach proposes to interface objects with relational database mechanisms and to achieve efficiency by caching copies.**

Software Architectures

40

Lecture 22
Architectural Construction Languages

Mary Shaw

Software Architectures ━━━━━━━━━━━━━━━━━━━━━━━━━━━━

Current Module Interconnection Languages

- **Assume module structure like Ada, Modula**
- **Support visibility control for names**
  - > provides / requires
  - > various granularities (entire modules to subfields)
- **Mostly support single kind of interconnection**
  - > usually procedure call
  - > others include data flow (UNIX shell)
  - > often support a single language
- **Most handle only access rights, type checks**
- **Can't take advantage of special properties**
  - > (e.g., an abstract data type is specified as algebra)
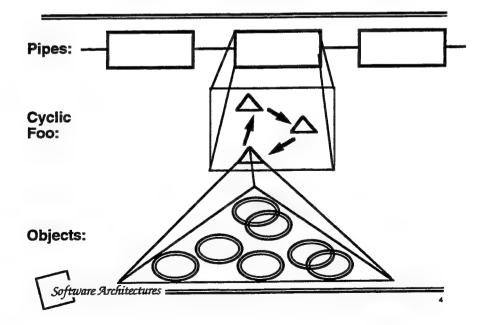
Software Architectures ━━━━━━━━━━━━━━━━━━━━━━━━━━━━

Problems with Current Systems

- **Inter-module connection by name matching**
- **Topology embedded in module definitions**
- **Poor abstractions for many relationships**
- **Pre-emption by built-in mechanisms**
- **Elaboration by single-point expansions**

Software Architectures

3

Provide Abstraction Capabilities

Pipes:

Cyclic Foo:

Objects:

Software Architectures

4

Critical Elements of Design Language

- **Components**
  - > Module-level elements, not necessarily compilation units
- **Operators**
  - > For combining design elements
- **Abstraction**
  - > Ability to name elements for further use
- **Closure**
  - > Named element can be used like primitives
- **Specification**
  - > More than just computational functionality
  - > Specs of composites derivable from specs of elements

Software Architectures

5

Module Interconnection Languages

- **Like any language:**
  - > Communication between project team members
  - > Checkable means of documenting structure
- **Unlike programming level:**
  - > Project management tool
  - > Design tool for overall system structure

Software Architectures

6

Module Interconnection Relations

- **Resources**
  - \> Atomic
  - \> Nameable: variables, constants, procedures, types
- **Components**
  - \> Purely grouping node
  - \> Subsystem node with driver
  - \> Actual code
- **Relations**
  - \> System/subsystem parentage
  - \> Upward propagation of provided resources
  - \> Controllable sharing among siblings
  - \> "Uses"
- **Very strongly hierarchical in organization**

Software Architectures

7

Structural/Functional Elements

- **Computation:** *simple in/out relations, no retained state*
  - \> mathematical functions, filters, transforms, transducers
- **Memory:** *(shared) body of persistent structured data*
  - \> data base, symbol table, file system, directory, array, hypertext
- **Manager:** *state and closely related operations*
  - \> abstract data type, resource manager, many servers
- **Controller:** *governs time sequences of others' events*
  - \> scheduler, synchronizer
- **Link:** *Transmits information between entities*
  - \> communication link, remote procedure call, UI
- **Command system:** *discrete, repeated, usually local, syntax-intensive manipulation of an entity*
  - \> editors, operating systems, menu systems

Software Architectures

8

Combining Subsystems

- **Mechanisms for connecting subsystems:**
  - > Procedure call
  - > Data streams
  - > Instantiation
  - > Data sharing (direct access)
  - > Message passing
  - > Implicit triggering
  - > Intermingled code
- **Interface protocols:**
  - > Calling sequences
  - > Addressing assumptions
  - > Formal protocols
  - > Shared representations

• Software Architectures ═══════════════════════

9

Creating Systems

- **Subsystems may be composite or primitive**
  - > Composites are like systems
    - » Large designs require structure
    - » They can be composed of subsystems
    - » Different organizations can be used at different times
  - > Primitives at the architecture level are programs at a lower level
    - » Roughly at scale of a module
    - » Built in conventional programming languages

Software Architectures ═══════════════════════

10

Primitive Elements

- **Primitive architectural elements are non-primitive programs**
- **At this point, programmer may choose from many paradigms:**
  - > Imperative   Backtracking
  - > Rule-based State machine
  - > Constraint   Table-driven interpreter
  - > Functional Dataflow
- **Mixing of programming paradigms may be restricted by implementation constraints**

Software Architectures

11

Requirements for Architectural Support

- **Decomposability and composability**
  - > of both components and specifications
- **Independence of elements**
  - > standalone definitions, structure defined separately
- **Exposed legacy of prior design**
  - > codified systematically, with engineering design help
- **Generality**
  - > large variety of heterogeneous structures; non-preemptive
- **Capability for analysis**
  - > consistency, performance, choice among alternatives

Software Architectures

12

Expectations for Specifications

- **Description**
  - > say what it is
- **Construction**
  - > say how to build one
- **Verification**
  - > determine whether implement. matches specification
- **Selection**
  - > guide selection among alternatives for implementation
- **Analysis**
  - > determine implications of specification
- **Automation**
  - > construct one from the specification

Software Architectures

19

Language Support for Architecture

- **Base language**
  - > Provide uniform support for rich set of connections
  - > Make explicit distinctions among different kinds of components
  - > Separate specification of structure from implementation

- **Intermediate language**
  - > Add abstraction constructs: support multiple patterns
  - > Support multiple languages

Software Architectures

1A
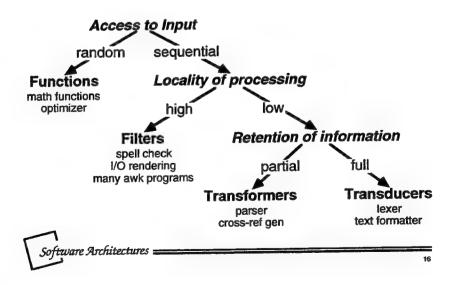
Language Support for Architecture (2)

- **Extended language**
  - > Add constructs for defining new abstractions:
    - » component types
    - » connection protocols
    - » configuration patterns
- **Graphical interface**
  - > Provide CAD-style interface

Software Architectures

Classifying Elements - Hierarchy
Computation Elements

Access to Input

random   sequential

Functions
math functions
optimizer

Locality of processing

high   low

Filters
spell check
I/O rendering
many awk programs

Retention of information

partial   full

Transformers
parser
cross-ref gen

Transducers
lexer
text formatter

Software Architectures
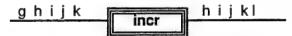
Classifying Elements by Property
Memory Elements
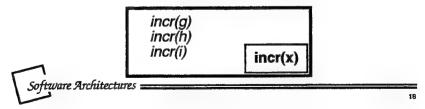
	Database	File Structure	Symbol Tab	Array
Model	various	hierarchical	associative	typed, passive
Duration	persistent	persistent	transient	transient
Stg Mgt	recoverable	buffer, free list	hash in fixed size	nil
Access	indexed	directory	hash	direct
Atomicity	per record	per file	non-issue	per scalar
Naming	key	paths	associative	index
Sharing	large	large	no	no
Capacity	large	large	small	small
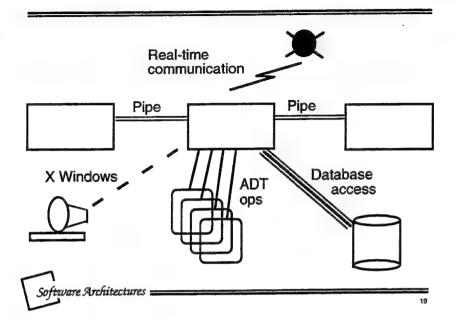
Software Architectures

17

Interface vs Base Functionality

- **Actual utility of a component depends on the way it's packaged as well as what it computes.**
- **Example:**
    - **> UNIX supplies same functionality packaged both as filters and system calls**
    - **> *Filter:* incremental processing on a stream**

g h i j k | incr | h i j k l

   - **> *System call:* one call per item, single thread of control**

```
incr(g)
incr(h)
incr(i)          incr(x)
```

Software Architectures

18

Abstractions for Connectors

Real-time
communication

Pipe | Pipe

X Windows

ADT
ops

Database
access

Software Architectures

19

Gap Between Tools and People

- **People describe designs in terms of abstract connections:**
    - > remote procedure call
    - > broadcast
    - > client-server
    - > pipe
    - > MIF, RFT, SYLK, ...
    - > event
- **Programming languages describe systems in terms of language constructs:**
    - > procedure call
    - > data export
- **When tools are this drastically mismatched, there are many opportunities for error**

Software Architectures

20

Problems with Current Practice

- Can't localize information about interactions
- Poor abstractions
- Poor structure for interface definitions
- Programming language specifications forced to do too much
- Poor discrimination of packaging differences and support for fixing mismatches
- Poor support for multi-language, multi-paradigm, or legacy systems

Software Architectures

21
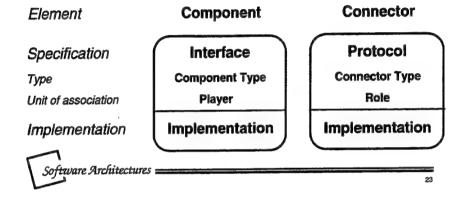
UniCon: Universal Connection Language

- Support common abstraction idioms
- Specify packaging properties as well as functional properties (how as well as what)
- Make connectors first-class
- Make abstraction mapping explicit
- Allow use of externally-developed tools

Software Architectures

22

UniCon: Universal Connector Language

- **Two major symmetrical constructs**
  - > Components: computation and data capabilities
  - > Connectors: mediate interactions among components

Element	Component	Connector
Specification	**Interface**	**Protocol**
Type	Component Type	Connector Type
Unit of association	Player	Role
Implementation	**Implementation**	**Implementation**

Software Architectures

23

Component Types Supported

- **Module (intuition: compilation unit)**
  - > Routine def & call, global data def & use, files
- **Computation (intuition : pure function)**
  - > Routine def & call, global data def & use
- **SharedData (intuition: Fortran common +)**
  - > Global data def & use
- **SeqFile (intuition : UNIX file)**
  - > Read next, write next
- **Filter (intuition : UNIX filter)**
  - > Streams in & out
- **Process (intuition : UNIX process)**
  - > RPC def & call
- **SchedProcess (intuition: real-time process)**
  - > RPC def & call, segment, trigger
- **General (intuition: anything goes)**

Software Architectures

24

Connector Types Supported
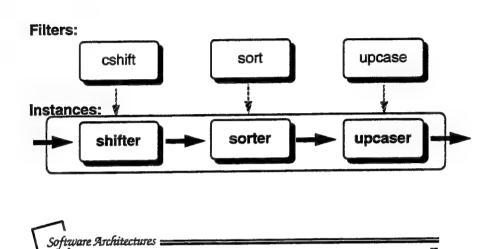
- **Pipe (intuition: UNIX pipe)**
  - > Source & sink
- **FileIO (intuition: UNIX ops between process & file)**
  - > Reader, readee, writer, writee
- **ProcedureCall (intuition: architectural use of proc)**
  - > Definer, caller
- **DataAccess (intuition: shared data within process)**
  - > Definer, user
- **RemoteProcCall (intuition: RPC)**
  - > Definer, caller
- **RTScheduler (intuition: processes compete for time)**
  - > Stimulus, action

Software Architectures

25

Connector Types Supported (2)

Software Architectures

26

"Given" Example

Filters:

```
[cshift]        [sort]        [upcase]
```

Instances:

```
→ [shifter] → [sorter] → [upcaser] →
```

Software Architectures   27

"Assigned" Example

Filters:

```
[finger]  [cut]  [diverge]  [cshift]  [converge]  [sort]
```

Instances:

```
                              [shifter]
→ [gather] → [prune] → [split]  ⤢  [merge] → [sorter] →
```

Software Architectures   28

Experience with a Course on Architectures for Software Systems   **20-14**

Provide Abstraction Capabilities

Pipes:

Cyclic Foo:

Objects:

Software Architectures

29

"Given" Example

Filters:

| cshift | sort | upcase |

Instances:

shifter → sorter → upcaser

Software Architectures

30

"Assigned" Example

Filters:

finger	cut	diverge	cshift	converge	sort

Instances:

gather → prune → split → shifter → merge → sorter

Software Architectures

31

Unicon Architecture

Unicon Source

Insert, delete

Librarian → **Parser** ← **Builder**

Build → ... etc ... / make / c compiler

Cxn Expert

Unicon source   (.vit)   Symbol table

Tree with file references   (.obj)

Connection expertise   (templates, library code, rules for each primitive connection)

Primitive Components   (.exe, .o, .c, ...)

Executable

32

Lecture 23
Connection Formalisms

David Garlan

Software Architectures ══════════════════════════════

Outline

- **The nature of architectural description**
- **The Wright specification language**
- **Connectors as protocols**
- **Properties of connectors**
- **Compatibility checking**
- **Some related work**

Software Architectures ══════════════════════════════

Modularization

- **Large systems require modularization to be manageable**
  - > intellectually & methodologically
- **Common approach to module description is based on definition/use relationships**
  - > describes organization of code
  - > induces a "depends-on" graph
  - > supported by module interconnection languages and programming languages
  - > good for the compiler
  - > tool support: type checkers
  - > lots of theory
- **But this is not the only useful form of modularization**

Software Architectures ==

3

Typical Descriptions of Software Architectures

- > "Camelot is based on the **client-server model** and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector 87]
- > "We have chosen a **distributed, object-oriented approach** to managing information." [Linton 87]
- > "The easiest way to make the canonical sequential compiler into a concurrent compiler is to **pipeline** the execution of the compiler phases over a number of processors." [Seshadri 88]
- > "The ARC network [follows] the **general network architecture** specified by the ISO in the Open Systems Interconnection Reference Model." [Paulk 85]

Software Architectures ==

4
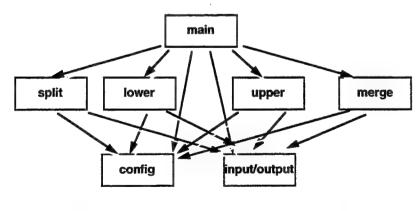
Example: Rule-Based System

5

Definition/Use Description

Produce alternating case of characters in a stream

Definition/Use Modularization

6

Architectural Description

```
                  ┌─────────┐
              ┌──▶ │  lower  │ ──┐
              │    └─────────┘   ▼
   ┌─────────┐                ┌─────────┐
───│  split  │                │  merge  │───
   └─────────┘                └─────────┘
              │    ┌─────────┐   ▲
              └──▶ │  upper  │ ──┘
                  └─────────┘
```

Software Architectures

7

Definition/Use versus Architectural

Definition/Use	Architectural
code modules	components/connectors
"uses" relationships	"interacts with" relationships
procedure call & shared data	pipes, client-server, event broadcast, ...
hierarchical reasoning	compositional reasoning
signatures	protocols
type checking	???

Software Architectures

8

The State of Architectural Description

- ## People do successful architectural design using
  - > ### Architectural styles and idioms
    - Pipes and Filters, Layered Systems, Client-Server Systems, Object-oriented Organizations
  - > ### Application-Specfic Frameworks
    - MacApp, Motif, Spreadsheets, Oscilloscopes,
- ## But these are usually
  - > Informal: box and line + prose
  - > Ad hoc: do what we did last time
  - > Un-analyzable: keep fingers crossed
  - > Un-maintainable: architectural drift
  - > Handcrafted: no tools

Software Architectures

9
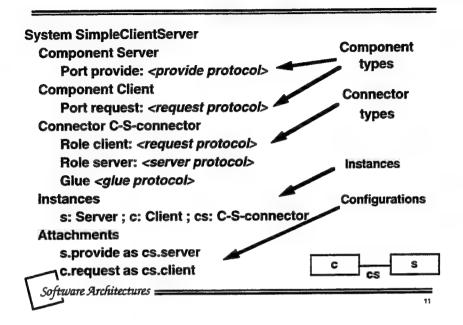
Formalizing
Architectural Representation
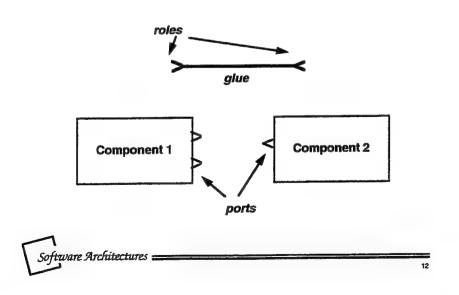
- ## Goal: Provide general formal model for architecture representation
- ## Approach:
  - > Formalize notion of boxes, lines, and configurations
  - > Describe connectors as first class entities
  - > Provide theory for reasoning about architectural descriptions

Software Architectures

10

Example: Wright

System SimpleClientServer
   Component Server
      Port provide: *<provide protocol>*
   Component Client
      Port request: *<request protocol>*
   Connector C-S-connector
      Role client: *<request protocol>*
      Role server: *<server protocol>*
      Glue *<glue protocol>*
   Instances
      s: Server ; c: Client ; cs: C-S-connector
   Attachments
      s.provide as cs.server
      c.request as cs.client

Component types
Connector types
Instances
Configurations

Software Architectures

11

Model of Connectors

roles

glue

Component 1   **Component 2**

ports

Software Architectures

12

A Formal Basis
for Architectural Connection

- **Notation (based on CSP):**
  - > **Events:** $e, f, push, pop, request, \sqrt{}$
  - > **Processes:** $P, Q, Stack, Client, Server,$ ✓
  - > **Prefix:** $e \rightarrow P, push \rightarrow Stack, request \rightarrow Server$
  - > **Internal Choice:** $P \sqcap Q$
  - > **External Choice:** $P \; [] \; Q$

Software Architectures

13

Example:
Specification of a Pipe Protocol

Connector Pipe
   Roles:

$Writer = (write!x \rightarrow Writer) \sqcap (close \rightarrow \checkmark)$

$Reader = Read \sqcap Exit$

$where \;\; Read = (read?x \rightarrow Reader) \; [] \; (read\text{-}eof \rightarrow Exit)$

$\qquad\qquad Exit = close \rightarrow \checkmark$

Glue $= \; Writer.write?x \rightarrow Glue \; []$

$\qquad\qquad Reader.read!y \rightarrow Glue \; []$

$\qquad\qquad Writer.close \rightarrow ReadOnly \; []$

$\qquad\qquad Reader.close \rightarrow WriteOnly$

$where \ldots$

Software Architectures

14

Specification of a Pipe Protocol (2)

$where \; ReadOnly = Reader.read?y \rightarrow ReadOnly$

$\qquad []$

$\qquad\qquad Reader.read\text{-}eof \rightarrow Reader.close \rightarrow \checkmark \; []$

$\qquad\qquad Reader.close \rightarrow \checkmark$

$and \; WriteOnly = Writer.write!x \rightarrow WriteOnly$

$\qquad []$

$\qquad\qquad Writer.close \rightarrow \checkmark$

Software Architectures

15

Connector Semantics

Connector C

    Roles: $R1 = R1; \; ...; \; Rn = Rn$

    Glue $= Glue$

is the (CSP) process:

    $Glue \parallel (R1: R1 \parallel ... \parallel Rn: Rn)$

where

    $n{:}P$ **labels all of the events except** $\sqrt{}$ **in**

        process P **with name** n

        and $Glue$ **alphabet is sufficiently large:**

        $\alpha \; Glue = R1{:}\Sigma \cup ... Rn{:}\Sigma \cup \{\sqrt{}\}$

Software Architectures

16

Connector Instantiation

- **Attaching a port as a role:**
  - > *Informally:* port stands in for the role.
  - > *Formally:* instantiating roles P1 .. Pn gives

    $Glue \parallel (\underline{R1}: P1 \parallel ... \parallel \underline{Rn}: Pn)$

- **Ports need not be identical to the roles**
  - > pipe can be connected to a file
  - > client can use a subset of a server's facilities
- **When is it ok to attach a port to a role?**

Software Architectures

17

Compatibility
(of a Port with a Role)

- <u>**Yes:**</u>
  - **Port** $P = (push \rightarrow P) \sqcap \checkmark$
  - **Role** $R = (push \rightarrow R) \sqcap (pop \rightarrow R) \sqcap \checkmark$
- <u>**No:**</u>
  - **Port** $P = (push \rightarrow P) \sqcap \checkmark$
  - **Role** $R = init \rightarrow R'$
    - <u>*where*</u> $R' = (push \rightarrow R') \sqcap (pop \rightarrow R') \sqcap \checkmark$

Software Architectures

18

Port-Role Compatibility

- **Informally:**
  - > A port P is compatible with a role R if the behavior of P does not violate the promises of R
- **Formally we can use process refinement:**

  $$"R \sqsubseteq P"$$

- **Details**
  - > Alphabets may not be the same

    Solved by augmenting alphabets of both processes.
  - > We only care about the behavior of the port in the context of the connection that it is participating in.

Relaxing Assumptions

- **To allow for greater opportunities for reuse**
  - > Do not want to insist on strict subset of behavior
  - > But want to make sure that port has required behavior in the context of use

- **So we use the following definition**

  P is compatible with R if

  $$R_{+(\alpha P \setminus \alpha R)} \sqsubseteq (P_{+(\alpha R \setminus \alpha P)} \parallel det(R))$$

  restricted to traces of R

Well-Formed Connectors

Property 1: A connector doesn't get 'stuck'.
> **Formally:**
> A connector is <u>deadlock-free</u> if whenever it reaches a state in which it cannot make progress, the last event to have been executed is $\sqrt{}$.

Property 2: The glue is truly a constraint on the behaviors of roles.
> **Formally:**
> A connector is <u>conservative</u> if
> $$traces(Glue) \subseteq traces\ (R1{:}r1\ \|\ ...\ \|\ Rn{:}rn)$$

Software Architectures

Reasoning about the Specifications

- Theory allows us to "compatibility check" architectural descriptions
- Analogous to (but subsumes) type checking
- Can be automated for this notation
- Guarantees important properties
  Theorem: (soundness)
  > If a connector is deadlock free and conservative then any compatible instantiation will also be deadlock free.

Software Architectures

Some Related Work

- **Other architectural description languages**
  - > Rapide – Luckham & Mitchell
  - > Unicon – Shaw
  - > ...
- **Protocol specification**
  - > Lotos, CCS, SML, etc.
- **"Regular types" – Nierstrasz**
- **Interaction categories – Abramsky**

Software Architectures

Lecture 25
Layered Architectures:
Network Protocols

Gregory Zelesnik

Software Architectures ═══════════════════
1

Layered Architecture

- **layering**
  - > the principle of collecting functions into related and manageable sets [PISC93]

- **layered architecture**
  - > a subdivision of the architecture of a system into layers of functionality
  - > a layer in such an architecture uses functionality in another layer
  - > a layer exposes functionality to another layer at the layer's interface (i.e., API)
  - > a service is implemented by a vertical slice of function invocations in one or more layers

Software Architectures ═══════════════════
2

Example 1: Stoneman

- **Stoneman (circa. 1982)**
  - > a DoD specification for Ada programming support environments (APSE)
  - > specified facilities:
    - » toolset that supports life-cycle development
    - » unified database for software objects
    - » extensible command language
    - » encapsulation of operating system services
  - > prescribes a layered architecture
  - > reflects philosophy:
    - » support for entire life cycle
    - » provide common background for programmers

Software Architectures

3

The Ada Language System (ALS)

- **Softech, Inc. (1985)**
- **Stoneman-compliant APSE implementation**
- **layered architecture**
  - > host operating system
  - > Kernel APSE (KAPSE)
  - > toolset

Software Architectures

4

The KAPSE

- a layer between the host operating system and the toolset
- provides toolset with a common interface to operating system services (e.g., as servers or library routines)
- provides developers with a machine-independent development environment
  - > same command language and access to same tools, regardless of host computer
- uses services in host operating system to implement its services

Software Architectures

Example 2: Computer Networks

- the philosophers analogy
  - > each person (layer) thinks s/he is communicating with her/his peer, horizontally
  - > actual communication is vertical between people (layers), except in layer 1
  - > the three protocols are completely independent
- networks are designed as layers:
  - > prevent changes in part of the design (a single protocol at a layer) from requiring changes of other parts
  - > promotes reuse of protocols

Software Architectures

The ISO OSI Reference Model

- **framework for describing layered networks**
- **discusses**
  - > layering
  - > uniform terminology
  - > seven layers, their purpose, functions, services
- **value of model -> it provides:**
  - > uniform terminology for network users and implementors
  - > generally agreed upon split of network activities
- **it is *not* a protocol standard**

Software Architectures

7

The Physical Layer

- **functions:**
  - > to allow a host to send a raw bit stream into the network
  - > transparent transmission of physical-service-data-units (bits) between data-link entities
  - > management of the physical layer services
- **lowest layer in the model**
- **only layer in which:**
  - > inter-machine communication actually occurs
  - > two or more machines are physically connected

Software Architectures

8

Services Provided

- **physical connections**
- **physical-service-data-units**
- **physical connection end-points**
  - > physical-connection endpoint identifiers
- **data-circuit identification**
  - > identifiers specifying data-circuits between systems
- **sequencing**
- **fault condition notification**
- **quality of service parameters**

Software Architectures ═══════════════════════
9

Physical Connections

- **physical media**
  - > twisted pair
  - > coaxial cable
  - > fiber optics
  - > line-of-sight (infrared, microwaves, radio)
  - > satellites
- **data-circuit (OSI)**
  - > a communication path in the physical media between 2 physical entities, together with the facilities necessary for transmission of bits on it

Software Architectures ═══════════════════════
10

Network Organization

- **a network is an interconnection of two or more systems to a physical medium**
- **end-to-end connection**
  > direct connection between 2 systems
- **multipoint connection**
  > several data-link entities share the same medium
- **network types**
  > local-area networks
  > long-haul networks

Software Architectures

11

Local Area Networks (LAN)

- **three characteristics:**
  > diameter of not more than a few kilometers
  > total data rate of at least several Mbps
  > complete ownership by a single corporation
- **example LAN configurations (IEEE 802)**
  > Ethernet (802.3)
  > Token bus (802.4)
  > Token ring (802.5)

Software Architectures

12

CSMA/CD (IEEE 802.3)

- 1-10 Mbps data transfer over various physical media (e.g., Ethernet)
- topology is the standard bus topology, or tree topology using repeaters
- tranceiver taps into the bus; a cable connects tranceiver and interface board
- protocol:
  > if host wishes to transmit, it listens to the cable
  > if busy, host waits for cable to go idle
  > host transmits (if collision, transmission stops; each host waits random time, retransmits)

Software Architectures

13

Token Bus (IEEE 802.4)

- 1, 5, and 10 Mbps data transfer over 75 ohm broadband coaxial cable (use for TV)
- topology is a bus topology that uses a logical ring topology
- hosts physically connected to a bus, logically organized into a ring
- protocol:
  > host gets permission to transmit from neighbor in ring
  > a single host at a time has permission to transmit (no collisions)

Software Architectures

14

Token Ring (IEEE 802.5)

- 1, 4 Mbps data transfer using shielded twisted pair
- topology is ring topology
- protocol:
  - > special bit sequence (token) placed on ring during idle by a host finishing a transmission
  - > a host removes the token from ring when it wants to transmit (only 1 host transmits at a time)
  - > each bit arriving at a host is copied off, then copied back onto the ring
  - > transmitting host removes its own bits from ring

Software Architectures
15

Long Haul Networks

- referred to as Wide Area Networks (WAN)
- typically span entire countries
- have data rates below 1 Mbps
  - > use the telephone system for data transmission
- owned by multiple organizations
  - > phone carrier owns the communications subnet
  - > organizations own the hosts
- LANs are preferable for local communication
  - > higher transmission rates (more bandwidth)
  - > lower error rates (>= 1000 times)
  - > simpler protocols

Software Architectures
16

WAN protocols

- **RS-232-C**
  - > specifies the meaning of each of the 25 pins on a terminal connector, and the protocol for transmitting data in analog form (waveform)
- **PCM (pulse code modulation)**
  - > specifies the translation of analog waveform signals to a digital representation
  - > samples waveform 8000 times per second
  - > digitizes amplitude of waveform in 8 bits
  - > transmits sample of waveform every 125 usecs
- **X.21**
  - > standard for emerging digital communication

Software Architectures

17

Physical Layer Protocols

- use of particular physical layer protocols depends on the physical medium and transmission technology used
- physical layer protocols are usually implemented in hardware:
  - > computer interface cards
  - > modems
- physical and data-link layers usually combine to implement a protocol
  - > e.g., in a token ring LAN, the token is a sequence of bits (a frame)

Software Architectures

18

Physical Layer API

- the interface between the physical and data-link layers is conceptually simple
  - \> establish connection
  - \> disconnect
  - \> receive bit
  - \> transmit bit
  - \> report collision
- interface between these layers is not always clearly physically delimited
  - \> at the data-link layer, the transmission technology gets extremely specialized

Software Architectures

19

The Data Link Layer

- functions:
  - \> organization of physical-service-data-units (bits) into frames, then data-link-service-data-units
  - \> transparent transmission of frames between network entities
  - \> error detection and recovery
    - » bits become garbled or lost during transmission
  - \> sequence control
- layer in which raw bits become organized into data
- uses services of the physical layer

Software Architectures

20

Data-Link Layer Services

- **data-link connections, connection IDs**
- **data-link-service-data-units (frames)**
- **sequencing**
  - > the order of frames is maintained across a data-link connection, but can be presented out of order to the network entity
- **flow control**
  - > network entity controls trans. rate of frames
- **quality of service parameters (per conn)**
  - > mean time between detected errors, service availability, transit delay, and throughput
- **channel allocation**

Software Architectures

21

Framing Techniques

- **character count**
  - > frame consists of a fixed-format header containing count of following characters
  - > problems: lost characters, changed counts
- **character stuffing**
  - > end-of-frame character
  - > problems: need escape character
- **bit stuffing**
  - > frames delimited by '01111110' sequence
  - > after five consecutive '1' bits in data, a '0' added
- **checksums (included in frame header)**

Software Architectures

22

Frame Transmission Protocols

- a.k.a. "flow control"
- stop and wait
  - > transmitting host waits for receiving host to acknowledge receipt before sending again
- sliding window
  - > transmitting host allowed to have multiple unacknowledged frames outstanding
- HDLC (high-level data link control)
  - > uses bit-stuffing
  - > address field for multipoint lines
  - > control field (seq nos, acks, line status info)

Software Architectures ━━━━━━━━━━━━━━━━━━━━━━━━
23

Channel Allocation (broadcast networks)

- slotted ALOHA (satellite networks)
  - > time slotted into fixed-length units
  - > similar to CSMA/CD, but no CS, and CD done differently due to 270 ms delay
- LANs
  - > CSMA/CD collision repair
    - » binary exponential backoff
    - » baton passing (similar to Token Bus)
    - » highest numbered contender
  - > ring network token repair
    - » host-monitoring, regenerating token
    - » slotted token

Software Architectures ━━━━━━━━━━━━━━━━━━━━━━━━
24

Data Link Layer API

- **primitives**
  - > wait
  - > ToNetworkLayer, FromNetworkLayer
  - > ToPhysicalLayer, FromPhysicalLayer
  - > StartTimer, StopTimer
  - > StartAckTimer, StopAckTimer
  - > EnableNetworkLayer, DisableNetworkLayer
- **types**
  - > packet, FrameKind, frame

Software Architectures

25

Network Layer

- **functions:**
  - > transparent transmission of network-service-data-units between transport entities
  - > masks the differences in transmission and subnetwork technologies
  - > routing and relaying of packets
  - > segmenting and blocking
  - > error detection and recovery
- **may contain IMPs** (point-to-point)
- **empty for broadcast networks**

Software Architectures

26

Network Layer Services

- **network addresses**
  - > means for uniquely identifying transport entities
- **network connections**
  - > provide transfer of data between transport entities using network address
  - > all point-to-point; > 1 allowed between 2 entities
- **network-service-data-units (packets)**
- **error notification**
- **sequencing**
- **expedited network-service-data-unit transfer**
- **quality of service parameters**

Software Architectures ================================
27

Routing

- **a frame arriving at an IMP is converted into a packet, then routed**
- **algorithms:**
  - > static directory routing
    - » IMP has table of outgoing lines, indexed by destination
  - > hot-potato routing; shortest queue plus bias
    - » packet assigned to outgoing line with shortest queue
    - » combination of static directory and hot-potato routing
  - > delta routing (with routing control center)
  - > distributed adaptive routing (early ARPANET)

Software Architectures ================================
28

Congestion

- **permits**
  - > similar notion to tokens, but network wide
  - > limits number of packets in network, but not number at a particular IMP
  - > problem: permit regeneration

- **choke packet**
  - > when line utilization > some trigger value, choke packet sent to all sources of packets for that line

- **discarding packets**
  - > favor those having made greater # of hops

Software Architectures

29

X.25 (A Network Layer Protocol)

- **X.25 definitions**
  - > a host is a DTE (data terminal equipment)
  - > carrier's equipment is a DCE (data circuit-terminating equipment)
  - > an IMP is DSE (data switching exchange)

- **X.25 describes layers 1, 2, and 3**
  - > physical layer -> X.21 or X.21 bis (RS232-C)
  - > data-link layer -> HDLC variant (LAP or LAPB)
  - > network layer -> description of managing connections between pairs of DTEs

- **virtual call vs. permanent virtual circuit**

Software Architectures

30

X.25 Connections

- **DTE A sets up a connection with DTE B:**
  - \> DTE A sends a CALL REQUEST packet to DCE
  - \> DCE delivers it to DTE B
  - \> DTE B sends a CALL ACCEPTED packet to DCE, which is forwarded to DTE A
  - \> DTE A receives packet as CALL CONNECTED packet, and connection is established
- **full duplex communication occurs**
- **either DTE disconnects:**
  - \> DTE A sends a CLEAR REQUEST packet
  - \> DTE B sends a CLEAR CONFIRMATION ack

Software Architectures

31

IP (Internet Protocol)

- **network layer in the ARPANET (Internet)**
- **connectionless**
- **datagrams are transparently dumped onto the network, transported to the dest. host,**
- **decision made as Internet grew; some networks were unreliable -> reliability mechanisms moved to transport layer**
- **transport layer breaks messages up into datagrams of up to < 64K bytes**
- **network layer adds an IP header**

Software Architectures

32

Network Layer API

- **ISO standard 8348: OSI Network Service Primitives**
- **connection-oriented**
  - \> N-CONNECT
    - » establishing connections
  - \> N-DISCONNECT
    - » releasing connections
  - \> N-DATA, N-DATA-ACKNOWLEDGE, N-EXPEDITED-DATA
    - » using connections
  - \> N-RESET
    - » resetting connections
- **connectionless**
  - \> N-UNITDATA
  - \> N-FACILITY
  - \> N-REPORT

Software Architectures

33

The Transport Layer

- **functions:**
  - \> transparent transmission of transport-service-data-units (messages) between session entities
    - » shield transport entities from network anomalies
      - lost packets
      - packets delivered out-of-sequence
    - » choose cost-effective transmission mechanisms
    - » provide network-layer-independent primitives
  - \> mapping of transport addrs -> network addrs
  - \> management of transport connections
  - \> end-to-end error detection and recovery
  - \> expedited transport-service-data-unit transfer

Software Architectures

34

Transport Layer Services

- **transport layer**
  - > highest layer concerned with data transfer over the network
  - > reorders out-of-sequence packets
  - > senses & terminates bad network connections, reestablishing new ones to continue transfers
  - > provides services to session entities at TSAPs
- **services**
  - > transport connection establishment
  - > data transfer
  - > transport connection release

Software Architectures

35

Connection Establishment

- **transport layer program: transport station**
- **connection establishment**
  - > obtain a network connection which matches requirements of session entity
  - > decide on optimizations for use of network entities
  - > establish optimum size of data passed to the network layer
  - > select functions to be active during data transfer
  - > map transport addresses to network addresses

Software Architectures

36

Data Transfer/Connection Release

- **data transfer**
  - > sequencing, blocking, concatenation, segmenting
  - > multiplexing, splitting, flow control
  - > error detection and recovery
  - > expedited data transfer
  - > transport-service-data-unit delimiting
- **connection release**
  - > notification of reason for release
  - > identification of transport connection released

Software Architectures

37

Transport Layer Primitives

- **connection-oriented**
  - > T-CONNECT
    - » establishing connections
  - > T-DISCONNECT
    - » releasing connections
  - > T-DATA, T-EXPEDITED-DATA
    - » using connections
- **connectionless**
  - > T-UNITDATA
- **differences between Transport/Network primitives**
  - > N-primitives are intended to model network, warts and all
  - > T-primitives are intended to provide error-free service

Software Architectures

38

Transport Layer API (Unix)

- **T-CONNECT**
  - > request   socket(), bind(), connect(), setsockopt()
  - > indication   return from accept(), getsockopt(), following socket(), bind(), listen()
  - > response
  - > confirmation   return from connect()
- **T-DATA**
  - > request   recv(), sendv()
  - > indication   return from recv(), sendv(), select()
- **T-EXPEDITED-DATA**
  - > request   sendv() with MSG_OOB flag set
  - > indication   SIGURG, getsockopt() with TPFLAG-XPD, return from select()
- **T-DISCONNECT**
  - > request   close(), setsockopt()
  - > indication   SIGURG, error return, getsockopt()

Software Architectures

39

TCP (Transport Control Protocol)

- **specifically designed to tolerate unreliability**
- **accepts messages from session entities**
- **breaks them up into segments < 64k bytes**
- **adds header; gives datagram to network layer**
- **reassembles packets received in wrong order**
- **well-defined service interface**
  - > primitives for actively, passively initiating conns
  - > send and receive data
  - > gracefully and abruptly terminate connections

Software Architectures

40
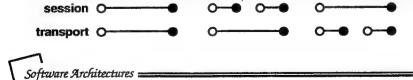
The Session Layer

- **functions:**
  - > provide for presentation-layer entities to:
    - » organize/synchronize their dialogue
    - » manage their data exchange
  - > session connection to transport connection mapping
  - > session connection flow control, recovery, and release
  - > expedited data transfer
- **presentation-layer entities are generally thought of as processes**

Software Architectures
41

Session Layer (cont.)

- **establishes and maintains connections, *sessions*, between pairs of processes**
- **hides details of transport protocols, transport addresses, from user processes**
- **session services maintain state of dialogue even over data loss by transport**
- **ways of mapping sessions onto transport connections (e.g., airline reservation system):**

session   O————————●   O——● O——●   O————————————●

transport O————————●   O————————●   O——● O——●

Software Architectures
42

Session Layer Services

- **data exchange**
  - > same type of connection protocol as transport
    - » establishment, data exchange, disconnection
    - » negotiated quality of service parms with peer
  - > orderly release of connection
    - » involves handshake protocol
    - » no loss of data
- **dialogue management**
  - > in principle, all OSI connections are full duplex
  - > applications may need half duplex
  - > session layer keeps track of whose turn it is to communicate (data token)

Software Architectures

43

Session Layer Services (cont.)

- **synchronization**
  - > used to move a session back to a known state (transport layer only masks comm. errors)
  - > e.g., teletex service
  - > session layer splits text into pages, inserts synch points - sending session must hold data
  - > major/minor synch points
- **activity management**
  - > user splits message stream into activities
  - > e.g., multiple file transfer
  - > e.g., phone banking transaction

Software Architectures

44

Session Layer Primitives

- S-CONNECT   Establish a session
- S-RELEASE   Terminate a session gracefully
- S-U-ABORT   User-initiated abrupt release
- S-P-ABORT   Provider-initiated abrupt release
- S-DATA   Normal data transfer
- S-TOKEN-GIVE   Give a token to the peer
- S-TOKEN-PLEASE   Request a token from the peer
- S-SYNC-MAJOR   Insert a major synch point
- S-SYNC-MINOR   Insert a minor synch point
- S-RESYNCHRONIZE   Go back to previous synch point
- S-ACTVITY-START   Start an activity
- S-ACTIVITY-END   End an activity

Software Architectures

45

Presentation Layer

- **functions:**
  - > perform generally useful transformations on the data before they are sent to session layer
    - » text compression
    - » data encryption
    - » virtual terminal protocols
    - » file transfer protocols
  - > session connection establishment/release
  - > data transfer
  - > negotiation/renegotiation of data syntax
  - > transformation of data syntax

Software Architectures

46

Philosophers Example

- messages from Philosopher 1 are converted to/from the layer 2 protocol (English or Dutch) to/from Swahili

- messages from Philosopher 2 are converted to/from Telugu

- same logical abstraction as converting data syntax by using compression or encryption

Software Architectures ============================ 47

Presentation Layer Notes

- text compression most often done in application

- encryption/decryption most often done in the transport layer or data link layer

- virtual terminal protocols:
    > ARPANET Telnet protocol (scroll-mode)
    > data structure model (page-mode)

- presentation layer has a set of primitives for establishing dialogues with peer entities

Software Architectures ============================ 48

Application Layer

- where "user" applications reside
- functions:
  - > provide application processes with means for accessing the OSI open systems environment
  - > identification of intended comm partners
  - > determination of availability of comm partners
  - > establishment of authority to communicate
  - > agreement on privacy mechanisms
  - > authentication of intended comm partners
  - > synchronization of cooperating applications

Software Architectures
49

Example Application Layer Apps

- electronic mail
  - > X.400 Message Handling System (MHS) protocols
  - > OSI Message-Oriented Text Interchange Systems (MOTIS)
  - > e.g., ARPANET Simple Mail Transfer Protocol
- public information services (telematics)
- file transfer, access, and management
  - > ftp, network block transfer (NETBLT)
- job transfer and management (CICS)

Software Architectures
50

Carnegie Mellon University
Software Engineering Institute

An Architectural Evaluation of User Interface Tools

Gregory D. Abowd, Rick Kazman and Len Bass

April 18, 1994

Architectures of Software Systems, Lecture 26
Master of Software Engineering Program
Carnegie Mellon University

1

Carnegie Mellon University
Software Engineering Institute

Outline

Introduction

Perspectives of Software Architecture

Reference architectures for User Interface (UI) systems

Modifiability for UI systems

Architectural evaluation of existing systems

Conclusions

Understanding Product Claims

"We have developed...user interface components that can be reconfigured with minimal effort."

"This [model] allows the UIMS to be simple and independent of the graphics software and hardware as well as the data representation used by the application program."

"Serpent...encourages the separation of software systems into user interface and "core" application portions, a separation that will decrease the cost of subsequent modifications to the system."

"This Nephew UIMS/Application interface is better that [sic] traditional UIMS/Application interfaces from the modularity and code reusability point of views."

3

Software Architectural Analysis

A way to assess a system's architecture with respect to non-functional quality attributes.

Relies on:

1. a common architectural notation
2. an analysis of quality needs
3. concrete benchmarks

Quality Attributes

Software engineering considerations, e.g.,

- **maintainability**
- **portability**
- **modularity**
- **reusability**
- **development efficiency**
- **performance**

Often called "non-functional" qualities

5

What is Software Architecture?

Reference models?

Idioms?

Connection languages?

Design in a suit?

Design in the large!

Obstacles for Architectural Evaluation

No common vocabulary

Tendency to create new terms/descriptions

Difficult to link architectures with development concerns

Emphasis on functionality

Little discussion of life-cycle support

7

Disclaimer

Architectures are not intrinsically good or bad!

It's all about context.
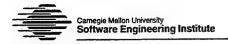
Perspectives on Software Architectures - 1

"[The software architecture level of design addresses] structural issues such as gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; composition of components; scaling and performance; and selection among design alternatives." [Garlan & Shaw]

9

Perspectives on Software Architectures - 2

"Architectural design is the activity of partitioning the requirements to software subsystems." [Sommerville]

Perspectives on Software Architectures - 3

"Software architecture alludes to two important characteristics of a computer program: (1) the hierarchical structure of procedural components and (2) the structure of data. Software architecture is derived through a partitioning process that relates elements of a software solution to parts of a real-world problem implicitly defined during requirements analysis." [Pressman]

11

Common Perspectives

- **Functional partitioning**
- **Structure**
- **Allocation**

Functionality

What the system does

Partitioned into conceptually simple pieces

Functional partitioning = domain analysis (in mature domains)

- **e.g., compiler: lexical analysis; parsing; code generation; code optimization**

In less mature domains, we use analysis techniques

- **object-oriented**
- **structured**

13

Carnegie Mellon University
Software Engineering Institute

Structure

Components (locus of computation)

- **filter, data store, object, process, server, *etc.***

Connectors (interactions between components)

- **procedure call, RPC, pipe, TCP/IP, *etc.***

Allocation
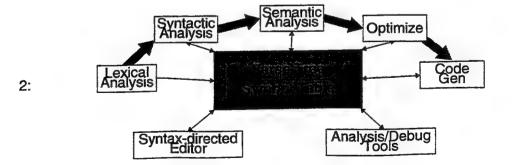
The mapping of functionality onto structure

Many different mappings are, in general, possible

15

Allocation: Example

1: Source → Lexical Analysis → Syntactic Analysis → Semantic Analysis → Optimize → Code Gen → Code

2:

Analysis of this mapping reveals the emphasis of the architecture.

16

User Interface Software Architectures

Reference architectures contain canonical information on all three perspectives.

- **Monolithic**
- **Seeheim**
- **Arch/Slinky**
- **PAC (presentation, abstraction, control)**

17

Monolithic
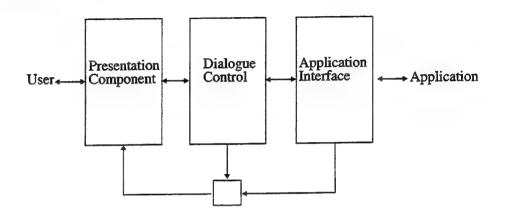
All functions in one structural component:

```
+---------------------------+
|                           |
|      Presentation         |
|                           |
|            +              |
|                           |
|      Application          |
|                           |
+---------------------------+
```

Seeheim
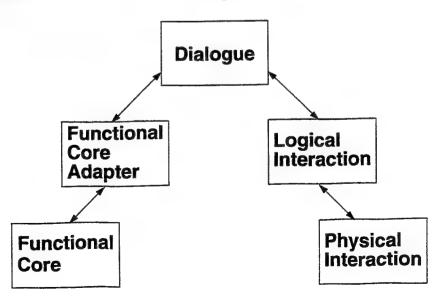
Introduced three functional roles

19

Arch/Slinky

Additional roles for further separation

20
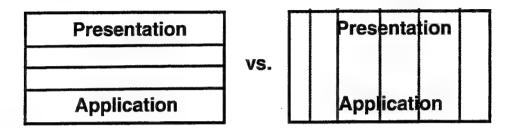
PAC

Slicing the pie another way

Presentation				Presentation		
Application	vs.			Application		
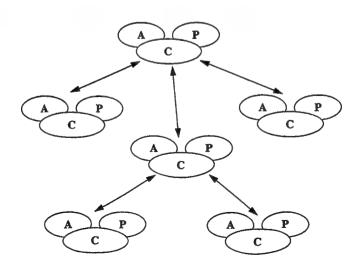
Separation vs. Rapid development

21

PAC

Hierarchical grouping of "vertical slices"

22

Modifiability for UI systems

Classes of modifiability:

- *Extension of capabilities*: adding new functionality, enhancing existing functionality;

- *Deletion of unwanted capabilities*: e.g. to streamline or simplify the functionality of an existing application;

- *Adaptation to new operating environments*: e.g. processor hardware, I/O devices, logical devices

- *Restructuring*: e.g. rationalizing system services, modularizing, optimizing, creating reusable components.

23

Choosing a Set of Concrete Tasks

In the User Interface domain, the most important types of modifiability are:

1. adaptation to new operating environments
2. extensions of capabilities

Benchmark tasks:

1. changing the physical interaction component, e.g. changing the toolkit
2. changing the dialogue, e.g. adding a menu option

Expressed most closely in terms of Arch/Slinky

Architectural evaluation of existing systems

Method

1. present developer's architecture
2. translate into a common language
3. compare with Slinky/Arch
4. evaluate for "performance" on benchmark modifications

25
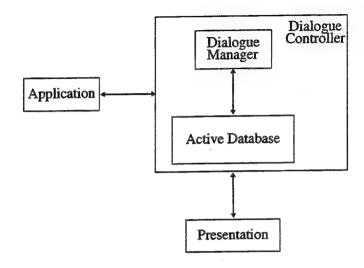
Structural Notation

We will use the following lexicon in this evaluation

Components		Connections	
▭	Process	(←)——→	Uni-/Bi-directional Data Flow
⬭	Computational Component	(◀)⟹	Uni-/Bi-directional Control Flow
▬	Passive Data Repository		
⬬	Active Data Repository		

Experience with a Course on Architectures for Software Systems   23-13
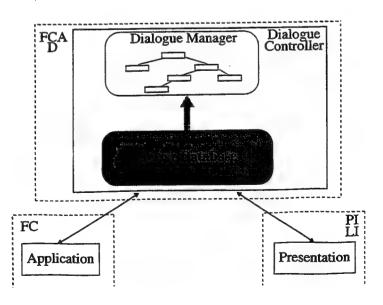
Serpent's Architecture

27

Serpent's Architecture (redrawn)

Experience with a Course on Architectures for Software Systems   23-14

Evaluation of Serpent

Task 1: separates PI/LI from the rest of the system but not from each other

- **some architectural support**
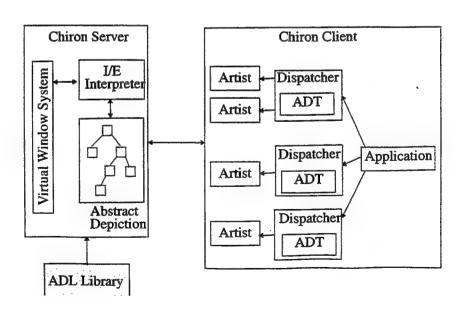
Task 2: Dialogue is a separate component, subdivided into view controllers

- **adequate architectural support for dialogue specification**

29
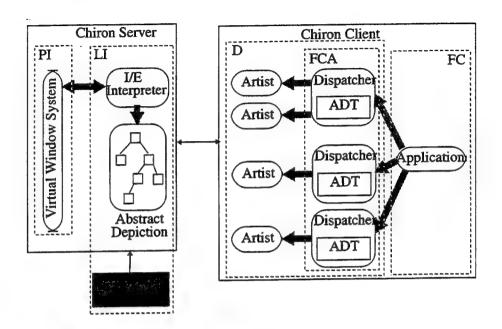
Chiron's Architecture

Chiron's Architecture (redrawn)

31

Evaluation of Chiron

Task 1: separates PI from LI and the rest of the system
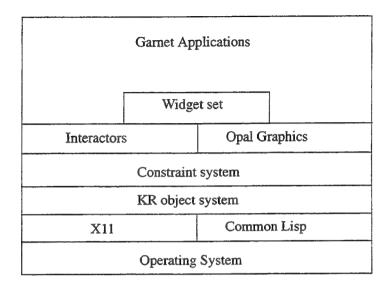
- **ideal architectural support**

Task 2: divides Dialogue between Artists and ADT/ Dispatchers

- **no clear guidelines for allocation of functionality**
- **some architectural support**

Garnet's Architecture

Garnet Applications
Widget set
Interactors	Opal Graphics
Constraint system
KR object system
X11	Common Lisp
Operating System

33
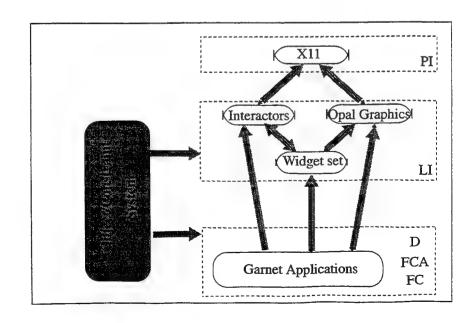
Garnet's Architecture (redrawn)

34

Evaluation of Garnet

Task 1: separates PI from LI and the rest of the system

- **ideal architectural support**

Task 2: dialogue is monolithic; extensive use of language features

- **no architectural support for dialogue specification**

35

Modifiability Evaluation: Summary

System	Task 1 Rating	Task 2 Rating
Serpent	medium	high
Chiron	high	medium
Garnet	high	low

Other Factors - 1

Example programs

Quality and volume of documentation

Ability to do rapid prototyping
- **compilation speed**
- **availability of builder**
- **interpretive**
- **use of data files**

Language issues

37

Other Factors - 2

Limitations/extensibility

Academic versus industrial

Ability to undo previous work

Completeness versus awkwardness

Learning curve/effects

Abilities of implementor

Conclusions

Architectural analysis can only proceed with:

- a common understanding of the various views of architecture

- a common representation of architectures

This method permits evaluation of an architecture in terms of an organization's life-cycle requirements

Architectures exist in a context—we must approach evaluation in terms of benchmarking

39

Lecture 27
Design or Default?
Decision Strategies for Software
System Design

Mary Shaw

Software Architectures

1

How Do You Choose an Architecture?

- **Organize the next system like the last one**
- **Adhere to company coding guidelines**
- **Follow the latest fad**
- **Use a prescriptive methodology or tool**
- **. . .**
- **Use the definitive architecture for your application domain**
- **Evaluate alternatives on the basis of**
  > **characteristics of the application requirements**
  > **constraints of the operating environment**

Software Architectures

2

Simple Design Space: Image Compression

100	**Bit image**	**Application: maps,**	
		engineering drawings	
Space **5**		**Group 3**	
factor		**raster**	
1		**Display**	**Group 4**
		list	**raster**

0.2   **2**   **20**

Display time (sec) for engineering drawing

Software Architectures

3
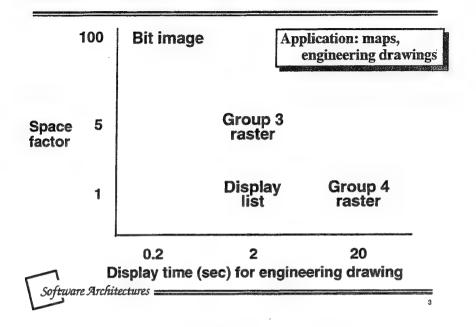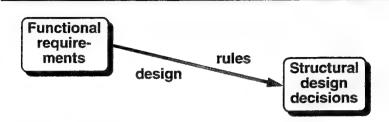
Image Compression: Special Considerations

- **Image characteristics**
  - > **Regularity improves compression factor**
- **Structural knowledge**
  - > **Display list has speed of Group 3, space of Group 4**
  - > **But you need to know where the lines are**
- **Tractability**
  - > **Often impractical to store full image**
- **Bandwidth interaction**
  - > **Bit image display may be limited by delivery bandwidth**
- **Latency and incrementality concerns**
  - > **Group 4 depends on context from previous (possibly distant) lines**

Software Architectures

4

Choosing a User Interface Architecture

- Design spaces for function and structure
- Some functional requirements favor or disfavor certain structures
  - > capture these as a set of preference rules
  - > develop prototype designer's advisor
- Similar problems exist for other architectural decisions

Software Architectures ================================
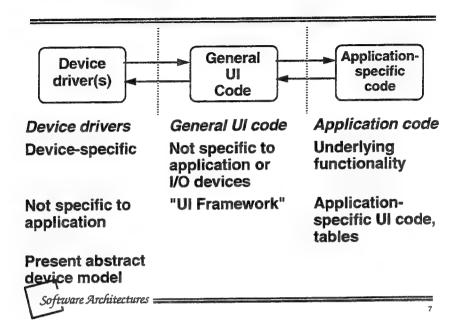
5

The Lane Strategy

- **Premises**
  - > More than one reasonable structure exists
  - > System requirements may favor or disfavor choices
- **Hypothesis**
  - > Design space can organize structural knowledge
  - > Design rules can (automatically?) guide structural choices
- **Experiment**
  - > Devise some rules
  - > See how they work on real systems
- **Results**
  - > Design space captures implicit knowledge in useful form
  - > It's worth trying to build a real design assistant

Software Architectures ================================

6

Basic Structural Model

```
┌──────────┐      ┌──────────┐      ┌──────────────┐
│  Device  │─────▶│ General  │─────▶│ Application- │
│ driver(s)│◀─────│   UI     │◀─────│  specific    │
│          │      │  Code    │      │    code      │
└──────────┘      └──────────┘      └──────────────┘
```

Device drivers

Device-specific

Not specific to application

Present abstract device model

General UI code

Not specific to application or I/O devices

"UI Framework"

Application code

Underlying functionality

Application-specific UI code, tables

Software Architectures

7

Functional dimensions

External event handling
    none   while awaiting input   preempt user

User customizability
    highmedium   low

User interface adaptability across devices
    none   local behavior   global behavior   appl semantics

Computer system organization
    uniprocess   multiprocess   distributed process

Basic interface class
    menu   forms   commands   nat'l lang   direct manip

Application portability across user interface styles
    highmedium   low

Software Architectures

8

Structural dimensions

Application interface abstraction level
    monolithic   abstr device   toolkit   [...managers...]

Abstract device variability
    ideal   parameterized variable ops ad-hoc

Notation for user interface definition
    [...implicit...]   [external, internal] x [declarative, procedural]

Basis of communication
    events   pure state   state & hints   state & events

Control thread mechanism
    none   HW proc   LW proc   non-preempt proc
           event handler   interrupt svc routine

Software Architectures

9

Design Rules

Basic rule format
    choice A [favors/disfavors] choice B with weight W
        (A, B from different dimensions)

Examples

high device bandwidth	+++	hybrid output comm basis
distributed system org	++	fixed or param fast input proc
distributed system org	+	event output comm basis
hi portability across styles	- -	hybrid output comm basis
implicit semantic info rep	-	coarse-grain appl comm

Rule set has about 170 such rules

Software Architectures

10

Validation Examples

Andrew Toolkit
  Environment for interactive graphical applications
cT (CMU Tutor)
  Programming language for computer-based instruction
Flight Simulator
  Prototype software for cockpit simulators
Genie
  **Pascal environment for novice programmers
    (Macintosh)**
Sage
  Automated graphical presentation of database queries
Serpent
  General-purpose user interface software substrate

Software Architectures
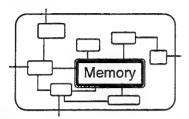
11

Notes on other slides

Slides summarizing Lane's validation experiments

Software Architectures

12

Evaluating Implementation Choices

Should you use a
database?
file system?
symbol table?
array?
hypertext system?
other?

- **What questions should you ask about the application?**
- **How do the answers help you choose an implementation strategy?**
  - > **how do you then make further design decisions?**
  - > **what classifications of the alternatives help?**
- **Can the memory questions be formalized?**

Software Architectures

13
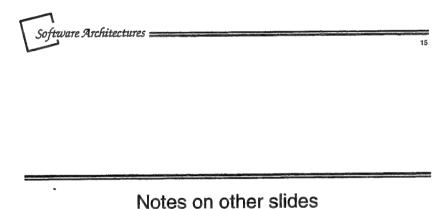
Quality Function Deployment

- **Quality assurance technique for translating customer needs into technical requirements**
  - > **List customer requirements & importance**
  - > **Identify implementation possibilities ("realization mechanisms"), with alternatives**
  - > **Decide how impl. possibilities relate to needs**
  - > **Determine interactions between impl. possib.**
  - > **Estimate difficulty**
  - > **Commit arithmetic**
  - > **Reexamine alternatives**

Software Architectures

14

Quality Function Deployment (2)

- **Makes requirements and rationale for implementation decisions explicit**
- **Many assignments of weights are highly subjective**
- **Arithmetic not well justified**

Software Architectures

15

Notes on other slides

Quality Function Deployment example

Software Architectures

16

Quantified Design Space

- *Developed by 1991-92 MSE studio group as merger of Lane's design spaces with QFD*
- Lane's design spaces
  - > Decomposition of design into dimensions
  - > Positive and negative correlations between dimensions
- QFD
  - > Relations between customer needs and realization mechan.
  - > Correlations between mechanisms

Software Architectures

17

Quantified Design Space (2)

- QFD framework captures design space knowledge, requirements, and requisite weights
- Massive arithmetic ensues
- Generates number for each design that can be compared to numbers for other alternatives

Software Architectures

18

Prospects for Design Guidance

- **Engineering design calls for selection based on needs of application**
    - > **Must have choices**
    - > **Must know what matters**
- **Accumulated experience should be accumulated in operational form**
    - > **For example, design rules**
    - > **Also need sources of alternatives (handbooks)**
- **Not very much guidance now exists in organized form**
- **Coping while we wait**
    - > **Informal guidance**
    - > **Rules of thumb**

Software Architectures

19

How Do You Choose an Architecture?

- **Organize the next system like the last one**
- **Adhere to company coding guidelines**
- **Follow the latest fad**
- **Use a prescriptive methodology or tool**
- **...**

Software Architectures

20

How Do You Choose an Architecture?

- **Use the definitive architecture for your application domain**
- **Evaluate alternatives on the basis of**
  - **> characteristics of the application requirements**
  - **> constraints of the operating environment**

Software Architectures

21

What is Software Architecture?

The papers:

> Garlan and Shaw (1993): Introduction to Software Architecture, Sec 1-3
> Perry and Wolf (1992): Foundations for the Study of Software Architectures
> Shaw (1993): Software Architectures for Shared Information Systems, Sec 1

Hints:

> The first four pages of the version of Garlan and Shaw (1993) in the course readings packet are out of order. If you number the pages as stapled 1 4 3 2 and read them in the order that implies, you will be able to make sense of them. Sorry 'bout that.

> Pay particular attention to Section 3 of Garlan and Shaw (1993) and to Section 1.3 of Shaw (1993). One of the primary objectives of this course is to sharpen your awareness of the decisions you make about software architectures and the assortment of different organizations from which you choose.

> We'll read all of the first and third papers in sections in the course of the semester. There's no harm in reading ahead.

Questions:

1) What does "design level" mean for software?

> This comes directly from Section 1.3 of Shaw (1993)

2) What are the *major* abstractions used by Garlan/Shaw and by Perry/Wolf to describe large systems? That is, what structure does each group impose on answers to a question such as "what's this architecture?"?

> Perry/Wolf:
>
> > Elements: processing, data, connecting elements
> > Form: properties, relationships, constraints on element
> > Rationale: motivation for the choices
>
> Garlan/Shaw:
>
> > System organizations: data abstractions; pipes and filters; layered systems; rule-based systems; blackboard systems
> > Components: different kinds of computational elements
> > Connectors: different ways of mediating the interactions among components
> > Constraints: restrictions on how a group of components and connectors can be combined

3) Name and briefly identify six architectural styles.

> These come directly from the subsections of Section 3 of Garlan and Shaw (1993).

Experience with a Course on Architectures for Software Systems　　　25-1

Classical Module Interconnection Languages

The papers:

> DeRemer and Kron (1976): Programming-in-the-Large versus
> Programming-in-the-Small
> R. Prieto-Diaz and J. M. Neighbors (1986): Module Interconnection Languages

Hints:

Read for the big ideas about system organization. Do not get bogged down in the notation of DeRemer and Kron, Section V, or the specific notations of the systems surveyed in Prieto-Dias and Neighbors.

Questions:

1) What important advantages of languages for programming-in-the-large versus languages for programming-in-the-small do DeRemer and Kron identify in their paper?

MILs give a "concise, precise, and checkable" model of program structure. They provide a way of communicating design and system documentation, and for managing a programming project. (And abstraction at multiple levels is generally a Good Thing.)

2) What kinds of analysis and checking do current MILs support?

Static type checking.

Interface matching:

- Access rights.

- Resource accessibility.

3) In Intercol, what is the distinction between a System Family and a System Composition?

A System Family is a set of different versions of the system. A System Composition describes the organization of the modules making up the System Version.

Information Hiding and Objects

The papers:

> Parnas, Clements, Weiss (1985): The Modular Structure of Complex Systems
> Booch (1986): Object-Oriented Development

Hints:

> Read for the ideas about system organization, not for programming details
> or prescriptions for the process of creating the systems.
>
> In a paper that you are not reading, Liskov pointed out that there are two
> major differences between abstract data types and "object-oriented" objects.
> The substantive difference is that objects include inheritance. The other,
> non-substantive, difference is terminology: different names for essentially
> the same things, such as *abstract data type/object, procedure/method,*
> *package/class,* and *procedure call/message.* You'll have to learn to put up
> with the terminology differences.

Questions:

1) What does each of the papers recommend as the primary criteria for decomposing
systems into modules?

> PCW: hide secrets: details that are likely to change independently should be separated; reduce
> likelihood that interfaces will change
>
> Booch: organize around objects in the real world

2) When a system contains large numbers of objects, some means of organizing them is
required. What does each paper propose? Which paper places most emphasis on this
aspect of the problem?

> PCW: module guide, strong hierarchy
>
> Booch: since each level is basically flat, create layers of abstraction

3) Contrast the provisions made by the two approaches for dealing with collections of
related definitions.

> PCW: nothing explicit, but hidden modules could provide shared information with limited
> distribution
>
> Booch: inheritance mechanisms allow new entities to be defined as variants of simpler existing
> entities

Modular Decomposition Issues

Garlan & Shaw　　　Questions and Answers on Readings for Lecture 5　　　Due: Mon. Jan. 24, 1994

The papers:

Parnas (1972): On the Criteria To Be Used in Decomposing Systems Into Modules.

Questions:

1) What does Parnas mean by "information hiding"?

Reveal as little as possible about the inner workings related to a design decision.

2) Give an example of information that is hidden in the second decomposition?

storage representations, algorithm for sorting, etc.

Formal Models

The papers:

> Shaw (1985): What Can We Specify?
> Spivey (1989): An Introduction to Z and Formal Specification
> Abowd, Allen, and Garlan (1993).
> > Using Style to Understand Descriptions of Software Architecture.

Hints:

In the Spivey reading pay most attention to how schemas are defined and combined. The material on refinements can largely be glossed over.

In Abowd, et al. you need not understand the details of the formalism, but you should try to understand the general form of the specifications, and the arguments at the end of the paper about why it is worth going to the trouble of producing the formalisms.

Questions:

1) Write a Z specification of the following system: a teacher wants to keep a register of students in her class, and to record which of them have completed their homework. Specify:

> (a) The state space for a register.

> (b) An operation to enroll a new student.

> (c) An operation to record that a student (already enrolled in the class) has finished the homework.

> (d) An operation to inquire whether a student (who must be enrolled) has finished the homework.

Answer: See attached answer.

2) What kinds of analyses and comparisons does a formal representation of style permit?

Specializations of a general style to more specific ones.

Comparison of auxiliary properties, such as hierarchical closure.

Check that syntactic constraints are consistent with the semantic equations: i.e., that all syntactically correct descriptions have a meaning.

Batch Sequential and Pipeline Systems

Garlan & Shaw   Questions and Answers on Readings for Lecture 7   Due: Mon. Jan. 31, 1994

The papers:

> Shaw (1993): Software Architectures for Shared Information Systems,
> Sections 2.introduction, 2.1, 3.introduction, 3.1, 3.2

Hints:

> This lecture begins the discussion of data flow systems. In such systems, the computation is dominated by the availability of data -- each component can execute only as fast as data is supplied to it. We will begin with two of the most common forms, batch sequential and pipeline systems (or pipe-and-filter architectures). The reading focuses on batch sequential systems. In addition, review your understanding of how to compose filters in UNIX; this is the handiest example of pipelines.

> You have already read Section 1 of this paper, and you'll read the rest for the lecture of March 14. The assignment for today is short, so you may want to read a few extra sections to get a little more context. You may also find it helpful to review sections 1-3 of Garlan and Shaw's Introduction to Software Architecture.

Questions:

1) What are the major components in a batch sequential data processing system?

> (1) an edit program; (2) a transaction sort; (3) a sequence of update programs; (4) a print program for periodic reports. [beginning of Section 2.1]

2) What are the major differences between batch sequential architectures and pipe-and-filter architectures?

> In a batch sequential system, each processing step may read all its input and run to completion before producing output. As a result, (1) the batch sequential system can use the input in arbitrary order, whereas a pipe-and-filter system does incremental processing; (2) batch sequential processing steps must be carried out in sequential order, whereas steps in a pipe-and-filter system can in principle execute concurrently.

3) What supports the assertion that a classical compiler is more like a batch sequential system than a pipe-and-filter system?

> The phases of a traditional compiler walk the parse tree in a complex order (not incrementally) and generally run to completion before passing control to the following phase. [paragraph 1, Section 3.2]

Experience with a Course on Architectures for Software Systems   25-6

Tektronix Case Study

The papers:

> Garlan and Shaw (1993): An Introduction to Software Architecture, Section 4.2.
> Garlan and Delisle (1990): Formal Specifications as Reusable Frameworks.

Hints:

> The purpose of the readings is to illustrate how Pipe & Filter systems can be applied in an industrial context. The formalism in the second paper is not the important part, although you should note how the form of the specification mirrors the form of the oscilloscope architecture.

Questions:

1) Why was a layered system rejected for the architecture of the system?

> Overall function of the system could not be naturally segmented into opaque layers of abstraction. For example, user interface must have access to acquisition setting.

2) The architecture adopted by the oscilloscope designers (as reported in Garlan and Shaw) departs from the general model of Pipe & Filter ways in at least two important ways. What were these?

> (a) Provided special inputs so user could configure the filter
>
> (b) Richer vocabulary of pipes: colored pipes.

3) Why was it felt necessary to introduce a richer vocabulary of pipes (i.e., "colored pipes") than is usually associated with a Pipe & Filter system?

> To get acceptable performance. For example, colored pipes allowed oscilloscope to avoid copying data, and avoided the problem of a slow filter holding back its upstream producer from supplying data to other filters.

4) List two ways in which the oscilloscope formed a "reusable framework" for Tektronix?

> (a) can substitute different components to get different oscilloscope functionality
>
> (b) can reuse the run-time code that supports interactions between filters even if completely change the set of filters used in a product

Pipe and Filter Implementation

The papers:

     Bach (1986): PIPES and DUP

Hints:

We previously covered the abstractions of pipes and filters. Now we turn to the classic actual implementation -- UNIX. The second assignment will involve creation of pipes, both directly and with shell commands. This reading and lecture are provided to help you learn practical plumbing.

Questions:

1) What abstract data type does a pipe implement? What common implementation of that abstract data type is used to implement pipes?

     (a) Queue of characters   (b) Circular queue or circular buffer

2) What is the difference between a named and an unnamed pipe?

     Named pipes opened with open system call, unnamed pipes with pipe system call

3) What happens when a process attempts to write an unnamed pipe for which there is no longer any active reader?

     Error signal (EPIPE, though the paper doesn't say so)

4) What must you do to guarantee that dup will assign to a specific file descriptor slot (for definiteness, say to file descriptor 3)?

     Make sure the lower-numbered slots are occupied -- dup assigns to the lowest-numbered free slot

Formal Models for Data Flow

The papers:

Allen and Garlan (1992): A Formal Approach to Software Architectures.

Hints:

You need not follow all of the formal details, but you should attempt to see if the model presented there matches your intuition about what a Pipe & Filter architecture is. Consider issues such as: How are schemas used? What is the essence of the model? You should also consider how this formal model differs from the one that you read about in the previous lecture.

Questions:

1) Contrasting this paper with the one by Delisle and Garlan that you read for the previous class, what is the essential difference between the two?

Delisle & Garlan provide a formal model of a specific system that uses a variant on the PF style, while Allen and Garlan provide a model of the PF architectural style itself.

2) List two ways in which the formal model of Pipes & Filters abstracts from reality.

Abstracts computation of filters as a state transition machine. Abstracts scheduling discipline. Abstracts notion of type. Abstracts internal state of filters.

3) List two constraints that the model places on the use of data flow?

Each port has at most one pipe connected to it. Types of data must match on either side of pipe. No dangling pipes. Filter must produce only valid internal states. Filter must produce only data of correct type. System must be started in a valid start state for each filter. Pipe can only be used once in a system description.

4) What key property relating to encapsulation does the model support? Why is it important?

Any subgraph of pipes and filters is equivalent to a filter. This allows one to describe a PF system hierarchically.

Communicating Process Architectures

The papers:

Andrews (1991): Paradigms for process interaction in distributed programs.

Hints:

Don't get caught up in the syntax of the programming notation. Read for the way in which problems are approached and for the qualities found in each problem and solution.

Questions:

1) What are the general techniques used in developing the solutions presented in the paper?

Client server, pipe filter, message passing (heartbeat, probe/echo) replicated data and computation, token passing.

2) What features of a problem would indicate that each of the following solutions might be suitable?

A. A heartbeat solution.

No global topology available, small diameter of net relative to number of processes, need to spread out local information globally, solution may require many iterations.

B. Replicated workers

Lots of spare processors, many unrelated tasks, no state shared between tasks

C. Probe/Echo

Depth-first algorithm is natural way to solve problem, tree structure of graph, solution can be calculated in single back-forth pass.

D. A network of filters

Computation is transformation on streams, can be calculated in incremental, functional steps.

Experience with a Course on Architectures for Software Systems   25-10

Formal Models of Processes

The papers:

Hoare: Excerpts from "Communicating Sequential Processes"

Hints:

Don't worry about any of the parts on "implementation".

Questions:

1) Write a CSP process that represents a student doing assignments for this course. Note there are four assignments, and each has a presentation associated with it.

    ASSIGNMENTS = ASSIGN1
    ASSIGN1 = (handin1 -> ASSIGN2) I (present1 -> handin1 -> ASSIGN2)
    ASSIGN2 = (handin2 -> ASSIGN3) I (present2 -> handin2 -> ASSIGN3)
    ASSIGN3 = (handin3 -> ASSIGN4) I (present3 -> handin3 -> ASSIGN4)
    ASSIGN4 = (handin4 -> STOP) I (present4 -> handin4 -> STOP)

Models of Event Systems

The papers:

> Garlan, Kaiser, Notkin (1992): Using Tool Abstraction to Compose Systems.
> Garlan and Notkin (1991): Formalizing Design Spaces: Implicit Invocation
> Mechanisms.

Hints:

> As usual, concentrate on the big ideas. For the first paper pay attention particularly to the argument about why ADT's have some serious limitations. The specific set of enhancements to KWIC are less important than the basic paradigm that they illustrate.

> For the second paper notice how Z is being used to provide both a general model that can be specialized for a particular system. Do not spend much time on the part of the formalism relating to the run time system

Questions:

1) Earlier in the semester we read two articles by Parnas, in which he advocated the use of information hiding and ADTs. What are the essential differences between that architectural style and the one advocated by Garlan, Kaiser and Notkin?

> Data encapsulated in ADTS versus data exposed to ``tools''.

> System functions invoked by explicit invocation versus triggered implicitly by data change events.

2) What are the tradeoffs in using one over the other?

> ADTs good if plan to change implementation.

> Toolies good if plan to augment function.

> Toolies not incompatible with ADTs, but expose more of structure than typically done with information hiding alone.

> Toolies require additional run time invocation mechanism.

3) According to the second article, in what way is Smalltalk's event mechanism flawed?

> There is only one event in the system: ``changed''. This makes it difficult for the receiver of the event (via the ``update'' method) to know what happened.

Architecture for Robotics

The papers:

Simmons (1993): Structured Control For Autonomous Robots.

Hints:

Approach this paper from an architectural perspective, asking yourself how the basic concepts map into the ideas that we have covered thus far.

Questions:

1) Characterize TCA architecturally:

a. What are the main kinds of components?

Task-specific modules.

The central control module.

b. What are the main kinds of connectors?

The different kinds of messages (command, query, constraint, etc.).

c. What are the main kinds of configurations (or topologies)

A star configuration, where the central control module is at the hub of the star.

2) Would you characterize TCA as an example of an Event System? Explain why or why not.

Yes and no. Yes because the architecture conveys the notion of implicit invocation. The sender of a message does specify its recipient. No because there can only be one recipient, i.e., no broadcast.

3) What is a task tree?

A structure which represents relationships between tasks and subtasks. Each relationship has one of three flavors: task decomposition, sequential achievement or delay planning.

Implementation of Event Systems

The papers:

Reiss (1990): Connecting Tools Using Message Passing in the Field Environment.
Notkin *et al.*(1993): Adding Implicit Invocation to Languages: Three Approaches.

Hints:

Both papers are concerned with techniques for implementing Event Systems, although Reiss refers to his system as a message-passing one. Pay attention to the way that each implementation leverages its operating domain: UNIX, in the one case, and specific programming languages, in the other.

Questions:

1) Pick four dimensions along which Field and one of the language-based implementations differ, and explain the differences?

Answers will vary, but here are some comparisons for the Ada implementation:

a. **Nature of Events:**

   Ada - static event declaration; Field - not clearly stated, but static by convention.

   Ada - either central or distributed; Field - not stated.

b. **Event Structure:**

   Ada - Parameters by Event Type; Field - Parameters by Event Type, but also arbitrary strings allowed.

c. **Nature of Event Bindings:**

   Ada - static; Field - dynamic

   Ada - selectable parameters; Field - same

d. **Announcement Syntax:**

   Ada - single procedure; Field not stated (but single procedure)

e. **What's a component?:**

   Ada - a package; Field - a process

Repositories: Blackboard Systems

Garlan & Shaw   Questions and Answers on Readings for Lecture 16   Due: Mon. Mar 7, 1994

The papers:

Nii (1986): Blackboard Systems, Parts 1 and 2

Hints:

First and foremost, read the Nii paper to understand the blackboard model and the kinds of problems for which it is appropriate. Study Hearsay and Hasp to see how the model is realized in two rather different settings, but don't get embroiled in fine details. Look at the other examples to see the range of variability available within the basic framework. Concentrate on the computational relations between the knowledge sources and the blackboard data structures. Notice the differences in control strategies, but -- again -- don't get bogged down in the details.

Questions:

1) In a few lines, describe the essential blackboard framework.

Blackboard has three major components: knowledge sources, blackboard data structure,   and control.

Knowledge sources partition domain knowledge; they contribute independently to   solving   the problem, can be represented in many ways, interact only via the   blackboard, and encode their conditions of applicability.

Blackboard data structures provide highly-structured hierarchical representation for   objects   that are intermediate and final results

Control provides opportunistic processing by monitoring blackboard changes.

2) What are three major differences between the nature of the processing required by the Hearsay-II and the nature of the processing required by the HASP system? (Note: this is a question about the processing requirements, not about the application domains.)

Data: Hearsay-II data given in advance (off-line), HASP data arrives continuously (on-line). Hearsay had input only at lowest level; HASP also had high-level input.

Computational style: Hearsay-II is motivated by generate-and-test, HASP by model-driven problem solving (relying on situation-specific knowledge). HASP could revise its conclusions but Hearsay could not; Hearsay, however, could hold multiple   hypotheses.

Time: In Hearsay-II, sequence of sounds in sentence; in HASP, the entire situation to be analyzed changes as time passes.

Completion: Hearsay-II attempted a correct interpretation of an utterance; HASP tried to make the best (partial) interpretation possible at the moment and improve it over time. Internal structure of blackboard. Differ in use of attributes and links. HASP also had   off-blackboard data.

Experience with a Course on Architectures for Software Systems   25-15

Databases and Client-Server Systems

The papers:

Gray & Reuter (1993): Selections from "Transaction Processing"
Mullender (1993): Selections from "Distributed Systems"

Hints:

From the readings try to develop your own precise definitions of terms such as "client-server system".

Questions:

1) To what extent does the UNIX file system satisfy the criteria for an adequate repository?

- **Complete**: No. The UNIX file system does not describe all aspects of a system, it just describes the files in it.

- **Extensible**: No. It is not possible to add new kinds of objects to the system. The only objects the UNIX file system can deal with are files, links, pipes, etc.

- **Active**: The UNIX file does not system provide mechanisms to automatically update properties of objects.

- **Local autonomy**: Yes. The UNIX file system can operate on its local objects (files), even though the site is disconnected from the others.

- **Fast**: Yes.

- **Secure**: Yes, (to some extent) the UNIX file system provides security mechanisms.

2) What is the fundamental difference(s) between the architectures portrayed in figures 1.9 and 1.10?

In figure 1.10 the resource managers have private locks and log managers, and the transaction manager does not provide an undo scan of the transaction log. It also assumes that the resource managers perform their own rollback.

3) Compare at-least-once versus at-most-once semantics. What property must requests have to make at-least-once semantics work?

At-least once protocols deliver messages once in the absence of failures, but may deliver messages more than once when failures occur. Such protocols work when requests are idempotent.

At most once protocols also deliver messages once in the absence of failures, but may not deliver messages at all when failures occur. Both parties in the communication must agree on the current protocol state, so that failures can be detected.

Evolution of Shared Information Systems

Garlan & Shaw   Questions and Answers on Readings for Lecture 18   Due: Mon. Mar 14, 1994

The papers:

>   [Shaw93] Shaw: Software Architectures for Shared Information Systems
>   [Eco93] The Economist: The Computer Industry
>   [Mor93] Morris & Ferguson: How Architecture Wins Technology Wars

Hints:

> This completes our coverage of this [Shaw93]. At this point most of the elements of the discussion should be familiar; review the sections we've already covered to be sure. The new ideas for today are the evolutionary progression and the appearance of a common pattern in several distinct application areas.

> [Shaw93] raises the issue of how heterogeneous systems should be integrated. addresses the market side of this issue -- how companies can position themselves (alone or in coalitions) to maximize their participation in heterogeneous markets. [Mor93] and [Eco93] examine the structure of the industry. As professionals you'll probably be interested in the whole papers, but for purposes of this course, focus on the discussions of standards and open systems and on the comparison of "old" and "new" industry structure.

Questions:

1) What is the common evolutionary pattern for shared information systems?

> (1) Isolated applications; (2) Batch sequential; (3) Repository; (4) Layered hierarchy

2) Compare the operational requirement on shared information in data processing applications with the requirements for software development environments..

> See [Shaw93] section 3.5, paragraph 1

3) What forces drove evolution from one architecture to another?

> Isolated applications to batch sequential: need to eliminate manual operations for regularly-used sequences of steps

> Batch sequential to repository: advent of on-line computing and need for interaction; also efficiency

> Repository to layered hierarchy: need to merge multiple repositories, especially with repositories distributed across many machines

4) Compare the "new" and "old" organizations of the computer industry. What is the significance of the shift?

> The old organization was based on vertical monopolies. The new organization reflects the effective market entry of many new companies, which requires their products to operate together. In the new organization the market is dominated by horizontal markets, each with several viable competitors

1) Briefly compare the advantages and disadvantages of open vs. proprietary systems.

> The question would be better-posed if it asked about open vs. closed systems. Nonetheless,.... Open systems provide a way to keep a market niche alive by encouraging other vendors to help produce enough products -- and enough prospects for future product development -- to keep market share. Closed systems, on the other hand, give the owning company a monopoly. On the down side, in an open system market your competitors are breathing hard down you neck and may consume you. But a closed system may not articulate with other systems, and the marketplace may be suspicious of systems with no second source.

Experience with a Course on Architectures for Software Systems   25-17

Interpreters and Heterogeneous Systems

The papers:

> [GS93] Garlan and Shaw: An Introduction to Software Architecture (the rest)
> [Wie92] Wiederhold: Mediators in the Architecture of Future Information Systems

Hints:

In GS93, concentrate on the ways the different system organizations are combined. Note that the original designers of these systems did not make the change of idiom explicit.

[Wie92] addresses the problems of making good use of large volumes of information from distinct independent sources, especially when it appears in multiple databases or supports multiple requirements.

Questions:

1) The earlier sections of this paper identify six classes of system organizations built up from smaller elements (both kinds of components and ways to connect those components). From your experience, extend these lists. Either add new kinds of organizations, components, and connectors or give some new substructure for the given classes.

> This question asks you to go beyond the readings and connect them to your own experience. There isn't a fixed set of additions or elaborations, but we'll try to distribute a summary of the interesting suggestions later.

2) Contrast Wiederhold's view of system construction using mediators with the view supported by a conventional module interconnection language (or the module connection mechanism of a conventional language.

> A conventional MIL has an essentially static view of system organization; the system designer enumerates explicitly the modules to use and the ways they are connected. Wiederhold's view, however, is essentially dynamic --- user applications dynamically identify appropriate mediators, which in turn dynamically select appropriate databases.

3) Section 4.5 of [GS93] examines the Hearsay-II blackboard architecture and shows how to separate abstract design concerns (the blackboard) from implementation concerns (the interpreter). Do the same for the HASP architecture (the other major example in the reading on blackboards).

> (See back)

Experience with a Course on Architectures for Software Systems   25-18

Newer MILs

The papers:

> Dewayne E. Perry (1987): Software Interconnection Models
> David Gelernter and Nicholas Carriero (1992): Coordination Languages and their
>   Significance
> Victor M. Mak (1992): Connection: An Intercomponent Communication Paradigm
>   for Configurable Distributed Systems

Hints:

In the Gelernter paper, there is a short description of Linda toward the end of the paper (page 103, leftmost column, third paragraph). Reading this paragraph first may be helpful to readers unfamiliar with Linda, though familiarity with Linda is not crucial to understand the author's main points.

Questions:

1) Explain how a coordination language such as Linda can be thought of as "gluing components together." Give a few examples of the "ad hoc" glue that is popular today.

Individual computations (processes, "executables") are often subsystems in a larger system (such as a distributed, parallel, or operating system). The glue that binds these systems together can be as rich in complexity and diversity as the systems themselves. The choices are many: is data or control exchanged? is the communication synchronous or asynchronous? what granularity of data? what are the guarantees? Because of this diversity, many idioms for data and control communication have arisen, e.g. RPC, message passing, fork and exec, barriers, file I/O, pipes and filters. Linda seeks to be a general "coordination language" that would replace all of these separate idioms with a general mechanism.

2) How does Mak's Connection paradigm allow software to scale?

The primary means of allowing software to scale is by supporting *composite* components, i.e. components that are composed of other components. Another way to view this is that today's component can be a sub-component of tomorrow's larger component. Whether a component is composite is abstracted away from the component's clients.

3) Briefly describe each of Perry's three models for connecting software components.

All three interconnection models (IMs) describe the *relationships* between *objects* (which can be pictured as a graph with labeled arcs); each model adds to the richness of the previous model by adding new objects and new relationships. The Unit IM describes relationships (such as "depends-on" and "includes") between large-grain objects (such as files and modules); Make is a good example of a system that conforms to this model. The Syntactic IM further describes the small-grain language constructs that compose the large-grain constructs (procedures, types, constants, variables) and their relationships ("is-composed-of", "imports", "exports", "calls"); the "classic" MILs conform to this model. The Semantic IM further describes the constraints for connecting the language constructs (preconditions, postconditions, obligations) and their relationships ("satisfies", "depends on", "propagated"); Perry's Inscape system conforms to this model.

Experience with a Course on Architectures for Software Systems   25-19

4) How do the module interconnection languages discussed earlier in the course (DeRemer and Kron, Prieto-Diaz and Neighbors) differ from the connection mechanisms in these readings? Would it be possible to combine an MIL with these newer connection mechanisms? (Why or why not?)

> Although, at the grossest level, both describe software units and the connections between them, they differ largely in domain. MILs describe modules, functions, types, constants, variables, etc. and the relationships between them, such as "is-composed-of", "imports", "exports", "calls", and "references." Systems like Linda and Connection, meanwhile, focus on whole computations (executables, processes) and the communication between them (data exchange, control exchange). Since these domains are complementary, combining them seems natural. A combined system would describe both the communication between computations, as well as the program units that make up a computation.

Interface Matching

The papers:

> James M. Purtillo and Joanne M. Atlee (1991): Module Reuse by Interface
> Adaptation.
> Brian Beach (1992): Connecting Software Components with declarative glue.
> Gordon S. Novak, Frederick N. Hill, Man-Lee Wan and Brian C. Sayrs (1991):
> Negotiated Interfaces for Software Reuse.

Questions:

1) What model does each of these three papers have about how components interact with each other?

> [Pa91] - Procedure call.

> [Bea92] - Software bus.

> [NHWS91] - Procedure call.

2) Briefly describe each of the three strategies for reconciling non-matching interfaces.

> [PA91] - Run-time transformation of parameters.

> [Bea92] - The software bus provides mechanisms for message transport and data sharing. The Software Glue Language provides data transformation.

> [NHWS91] - Code that performs run-time transformation of parameters. This code is generated using a menu-driven tool to specify the type and structure of data and subroutine interfaces.

3) In what ways (not restricted to those in today's readings) can component interfaces fail to match exactly but still be fixable?

> Parameters may not match (number, order, type).

> The representation of data may not match (ASCII vs. EBCDIC, little endian vs. big endian, units, etc.).

Connection Languages

Garlan & Shaw   Questions and Answers on Readings for Lecture 22   Due: Mon. April 4, 1994

The papers:

> Shaw & Garlan 93: Characteristics of Higher-level Languages for SW Architecture
> Shaw 94: Procedure Calls are the Assembly Language of Software Interconnection
> Shaw etc. 94: Abstractions for Software Architectures and Tools to Support Them

Hints:

All three papers address the same question: how can we provide notations and tools that are better matches to the ideas and vocabulary that real system designers actually use? Pay particular attention to the shortcomings of current systems, to why this can be treated as a language problem, and to the model of what a language should look like ("Abstractions and Tools" supersedes "Assembly Language" on this point). Cruise through enough of the Unicon description to answer the questions and see what it might do for you, especially the example that closely resembles one of your earlier assignments.

Questions:

1) The first part of the semester was devoted to describing architectural design idioms. When it comes time to implement these designs, real tools must be used. What deficiencies in these tools do today's readings describe? Can you name other deficiencies?

> This is the content of section 2 of "Assembly Language":
> Inability to localize information about interactions
> Poor abstractions
> Lack of structure on interface definitions
> Mixed concerns in programming language specification
> Poor support for components with incompatible packaging
> Poor support for multi-language or multi-paradigm systems

2) What are the essential elements of a computer language, whether it be for conventional programming or for architectural description?

> Components, operators, abstraction, closure, and specification

3) Identify those essential elements in the UniCon language.

> Components: Primitive components, primitive connectors, composite components

> Operators: Composition instructions, establishing associations between roles and players

> Abstraction: Ability to localize definition of a composite and give it a specification and a name

> Closure: Rule that allows composite component to be used as if primitive. Note that UniCon does not have operators that construct connectors or a closure rule for them -- but the model calls for this to be added.

> Specification: Attributes provided in property lists that can be used for checking compatibility either directly or with an external tool.

Experience with a Course on Architectures for Software Systems   25-22

Connector Formalisms

The papers:

Allen and Garlan 94: Formalizing Architectural Connection

Hints:

You might find it helpful to review your notes on CSP from Lecture 12 -- particularly insofar as they explain the distinction between CSP's deterministic (external) and non-deterministic (internal) choice operators. In approaching Wright, get a feel for the way protocols are decomposed using the notation. Also ask yourself what kinds of benefits the theory is buying you.

Questions:

1) In Wright, what are the parts of a connector description, and what function does each perform?

Roles describe the obligations of each participant in the interaction.

The Glue mediates the relationship between the roles and explains how the events of the roles are coordinated.

2) The paper talks a lot about deadlock freedom. Why is this a significant issue for understanding connection?

A deadlocked connector is one in which the various roles and glue do not agree on the joint behavior of the interaction: the communication gets stuck because some party can't make progress.

3) Explain informally what it means for a port to be compatible with a role? Why is this check important to perform?

The port lives up to the obligations of the role. This is important to check because otherwise we would have no guarantees that an instantiated connector behaves in a proper way -- i.e., it may deadlock.

4) Both Wright and UniCon have much to say about the nature of architectural connection. Outline the similarities and differences (if any) in the two approaches?

Answers may vary, but should include

Both view connectors as "first-class" entities that require their own specification.

Both use the idea of "protocol" to capture the notion of connector behavior.

Wright focuses on a particular formalism for specifying that behavior, while UniCon admits of different kinds of protocol specifications for describing connector behavior.

Layered Architectures:
Computer Networks

The paper:

> Tanenbaum, Andrew S. "Network Protocols," *Computing Surveys*, Vol. 13, No. 4, December 1981

Hints:

The article is long and full of detail about almost every aspect of the functions pertaining to each layer of the OSI Reference Model.

When reading it, focus on:

- the primary purpose of the OSI reference model itself
- the purpose of each layer
- the functionality/services provided by each layer

It is not necessary to focus on details such as header formats, bit sequences, etc. (e.g., the sliding-window frame transmission protocol: It is sufficient to know that this protocol allows a sender to have multiple unacknowledged frames outstanding during frame transmission. You do not have to know the exact algorithm)

Questions:

1) What is the primary reason for why networks are designed as a series of layers?

The primary reason for organizing networks into a series of layers is to group related functionality:

    a) to make the entire function of communicating simpler to implement and maintain

    b) to prevent changes in one part of the design from requiring changes in other parts

    c) to abstract away differences in technology that affect a small part of the design

    d) to reuse implementations of parts of the communication functionality

2) What is the OSI Reference Model? What is it not?

The OSI Reference Model is a framework for describing layered networks. It discusses the concept of layering in general and defines a uniform set of terms for network implementors and users to be able to discuss the various entities involved. Additionally, it describes the organization of functionality of network architectures into seven layers, and for each layer gives its purpose, the services provided to the next higher layer, and a description of the functions the layer must perform.

The value of the model is that it provides a uniform nomenclature and a generally agreed upon way of splitting the various network activities into layers.

The ISO Reference Model is not a protocol standard. It suggests places where protocols could be developed, but the standards themselves fall outside the domain of the model.

Experience with a Course on Architectures for Software Systems   **25-24**

3) You graduate from college and decide to keep in touch with one of your classmates. You both decide that you will write letters, but that you will only communicate via post cards. To save space you both decide to leave all English articles (e.g., "a", "the") out of the letters because their placement in the text is fairly obvious. Letters typically require more than one post card, so you number them in sequence starting at 1. When you've finished a letter, you mail all of the post cards at once. Also, you currently live with another classmate who is a good friend of the person with whom you are corresponding.

Assuming this is an analogy for communicating using a layered set of protocols, identify as many layers of the OSI reference model as you can that apply to this example. If you can, try to associate specific protocols for each layer from the reading with this contrived scenario as well.

This scenario is an analogy for communicating using a layered architecture. The analogy holds for arguably 5 of the 7 layers described by the OSI Reference Model. Even though the people communicating may not actually do this, think about them actually performing a series of translation steps on the ideas they generate. In other words, rather than immediately writing on post cards, think about them writing a letter first, copying it over without the articles, then copying it onto post cards, etc. Here is how the analogy holds for each of those 5 layers.

The Application Layer

The application layer in this analogy consists of the three classmates who are communicating. The two roommates are each communicating with their classmate that does not live with them. The classmate sometimes writes a single letter to be enjoyed by both roommates, and sometimes writes them individually. Each roommate writes letters to the classmate individually, and they may cooperate to jointly write a single letter to the classmate.

Sending: The classmates write their letters down on a piece of paper.

Receiving: The classmates read the letters given to them.

The Presentation Layer

The letters are read /written using a data (de)compression technique.

Sending: The classmates copy their original letter over, removing the articles (i.e., "a" and ("the").

Receiving: The classmates copy the received letter over , inserting appropriate articles where it makes sense to do so.

The Session Layer

The letters are actually sent and received.

Sending: One person in the house (in the case of the roommates) is given the responsibility of sending and receiving the letters. As a session layer activity, the letter is handed to this designated person.

Receiving: The designated person is given a copy of the letter. In the case of the roommates, a copy of the letter is given to the 2nd roommate by the designated person.

The Transport Layer

The letters are translated to and from post cards.

Sending: The designated person copies the letter onto post cards, numbers them, addresses them, stamps them, and places them in a U.S. Postal Service mailbox.

Receiving: The designated person collects the post cards, waiting until they all come in. This person organizes them in ascending order (they are often received out of sequence). When all post cards for a letter have been received, this person copies the post card letter onto a single piece of paper.

The Network Layer

The U.S. Postal Service delivers the post cards, one at a time, to their destinations. They may or may not use different routes and delivery mechanisms for each post card. Post cards may or may not be delivered to intermediate post offices when moving from the source post office to the destination post office.

Design Guidance

Garlan & Shaw   Questions and Answers on Readings for Lecture 27   Due: Wed Apr. 20, 1994

The papers:

Lane (1990): Studying Software Architecture through Design Spaces and Rules
Asada et al (1992): The Quantified Design Space

Hints:

Lane studied user interfaces for the purpose of organizing information about design decisions. For our purposes, the important parts of his work are the use of a taxonomy of characteristics to create a design space and the development of rules that relate characteristics of the problem to the architectural design decisions. Read the paper with this in mind. In particular, do spend enough time on the details of user interface structures to see what is going on, but do not spend an inordinate amount of time on the details. Asada, Swonger, Bounds, and Duerig were MSE students in the 1991-2 studio. After studying Lane's paper in this course in 1991 and Quality Function Deployment somewhere else, they applied these ideas to their studio project.

Questions:

1) What is a "design space"?

Design involves making choices among alternatives. Often you must make a number of decisions, each about selection from a set of choices. It is useful to think of having a multi-dimensional space, with one dimension to each set. Then a design can be thought of as a point in this space. By the way, a design methodology can then be thought of as a search strategy for the space -- and a good design methodology as one that leads you through portions of the space that are reasonably dense in acceptable solutions to your problem.

2) At various points in this course we have discussed -- quite informally -- suggestions about what circumstances might lead you to choose or avoid certain architectural idioms. Take the set of idiomatic patterns for system organization (objects, pipelines, events, blackboards, hierarchies, etc.) as one dimension of a target space (playing the role of Lane's structural space). Suggest some rules that might go into a design tool to help a designer make decisions along this dimension. What dimensions of the problem space (playing the role of Lane's functional space) do your rules suggest? In other words, use this opportunity to summarize some of the "rules of thumb" for selecting architectures that we've discussed in the course of the semester.

- If you can describe a solution for a computing engine that does not exist, consider an interpreter to provide a virtual machine.

- If a major part of the problem description involves coherent, independent collections of data structures and operations on those collections, consider encapsulating those data structures as objects.

- If the problem is primarily computational, and if it can be decomposed into a sequence of functions to perform on a stream of data, consider pipes.

- If the problem involves interpretation of complex data with considerable uncertainty (especially in a signal-processing domain), consider blackboards.

Experience with a Course on Architectures for Software Systems   25-27

These rules deal primarily with the way the computation of the problem is organized.

3) Consider the results delivered by Lane's system and by QDS. In each case you set up descriptions of the design alternatives. However, there are significant differences in the information they deliver, and consequently there are significant differences in the way the designer will interact with the two systems. Describe the major difference(s).

Lane's system evaluates the alternatives all at once and delivers rankings for the most promising few systems, whereas the user of QDS selects manually the alternatives to be examined -- that is, Lane's system explores the space whereas QDS allows the designer to sample it. On the other hand, with QDS it is much easier than with Lane to examine specific alternatives and to change the model. Additionally, Lane's design spaces are intended to serve many applications in one domain, whereas QDS' ancestry in QFD shows up in the creation of new spaces for each new application

Assignment 1
KWIC Using an Object-Oriented Architecture
Due: Wednesday, February 9.

1   Description of the problem

This assignment is to implement an interactive version of the KWIC index system (described in Parnas's *On the Criteria To Be Used in Decomposing Systems into Modules*) in the object-oriented paradigm. You will be provided with a partial Ada implementation of the system and asked to identify and make the necessary modifications.

The provided system is simply a line alphabetizer. It interactively inputs a line at a time and upon demand outputs an alphabetized list of the current collection of lines. Here is a transcript of a sample session:

```
Add, Print, Quit: a              Add, Print, Quit: a
> O my son Absalom               > O Absalom
Add, Print, Quit: a              Add, Print, Quit: p
> my son my son                          O Absalom
Add, Print, Quit: a                      O my son Absalom
> and the king cried                     and the king cried
Add, Print, Quit: a                      in a loud voice
> in a loud voice                        my son my son
Add, Print, Quit: p              Add, Print, Quit: q
        O my son Absalom
        and the king cried
        in a loud voice
        my son my son
```

Your assignment is to modify the existing code to support the following changes:

- Rather than simply outputting an alphabetic list of all the lines, the *Print* command should output an alphabetic list of the circular shifts of all the lines. However, shifts (including the nullary shift) which result in a line beginning with a trivial word—*a, an, and, the*, or the capitalized versions of these words— should be omitted.

  3 different ways of displaying should be supplied. Upon entery of a p at the command line, the system should print another "menu line":

```
Add, Print, Quit: p
Simple, Aligned, Concordance:
```

1. *Simple Display:*   Upon entery of a s at the command line the system should print the alphabetic list of all line shifts, as described above.

2. *Aligned Display:* Instead of printing the shifted line, the original sentence is printed, but the word that is at the begining of the shifted sentence is aligned, and capitalized. For example, the line "and the king cried" is printed as follows:

```
and the king CRIED
        and the KING cried
```

3. *Concordance Style Display:* For each shifted variant of the sentence the original sentence is printed, but the word at the beginning of the shifted variant is abbreviated. The line "and the king cried" is printed as follows:

```
and the king c.
and the k. cried
```

- three commands, *Original, Delete* and *Count* , should be added.

  1. *Original:* Upon entry of an o at the command line, the system should output a list of all lines entered by the user (including lines beginning with trivial words, but not including the circular shifts of lines) in their original order.

  2. *Delete:* Upon entery of a d at the command line, the system should prompt for a line (much like the Add command). This line should then be deleted from the system.

  3. *Count:* on entery of a c at the command line, the system should display the number of original lines currently in the system.

Here is a sample session of the new system:

```
Add, Print, Original, Delete, Count, Quit: a
> and the king cried
Add, Print, Original, Delete, Count, Quit: a
> in a loud voice
Add, Print, Original, Delete, Count, Quit: p
```

```
Simple, Aligned, Concordance: s
        cried and the king
        in a loud voice
        king cried and the
        loud voice in a
        voice in a loud
Add, Print, Original, Delete, Count, Quit: c
        2 lines
Add, Print, Original, Delete, Count, Quit: d
> and the king cried
Add, Print, Original, Delete, Count, Quit: c
        1 lines
Add, Print, Original, Delete, Count, Quit: p
Simple, Aligned, Concordance: c
        i. a loud voice
        in a l. voice
        in a loud v.
Add, Print, Original, Delete, Count, Quit: o
        in a loud voice
Add, Print, Original, Delete, Count, Quit:q
```

2   The current system

The current system is decomposed into the following modules:

- words,

- lines,

- line_collections, and

- tree_binary_unbounded_managed.

In addition there is a top-level module (session) which provides the command-line interface.

The source code for the current system will be made available to you by next class period. Watch the class bulletin board for instructions on how and where to obtain the code, along with instructions on accessing an Ada compiler.

3 Discussion

On Wednesday, February 2, two or three teams will briefly present their initial designs for class critique and discussion. Volunteers for this presentation will be solicited during the previous class period. Note that each team will be responsible for one such presentation over the course of the three assignments.

This presentation/discussion will not be graded. It is solely for the benefit of you and your classmates.

4 Due date and electronic hand-in

The assignment is due by 10:30 am on Wednesday, February 9. You should create a directory called "sa" in one of the team members home directory, and a subdirectory called "hw1". In "sa/hw1" prepare a file called "kwic.doc". This file should contain:

- the names of both team members,

- a list of the modules added/modified and for each such module a list of the resources added/modified.

The directory should also contain the system (source files, especially of all modules modifies/added, and an executable file, named "session").

You should email a pointer to that directory (machine name, user name) to the Teaching Assistant by the due date. After that, none of the files in the directory should be touched. In addition there will be a written commentary (due at the beginning of class on Wednesday, February 9) answering the following questions:

1. Describe the architecture of your system (both the provided part and the parts you added), explaining how it is an example of an object-oriented architecture, and in what ways (if any) it deviates from the basic object-oriented style. For each of the new functionalities required, descibe how your system implements it. Justify your design.

2. For each of the changes you made, explain if the change was of the internals of one of the system components (data structures or algorithms) or of the system architecure.

3. What changes would you have to make to your system to change the representation of line storage? What other components would be affected?

4. What changes would you have to make to your system to add the functionality of only showing lines that start with a particular word?

5. Does the system lend itself to a distributed implementation? If so what changes would have to be made to make it function this way?

The commentary should be your own work: i.e., individuals, not teams for commentary.

5 Grading criteria

Your solutions and commentary will be graded by the following criteria:

- Whether or not the resulting system performs as required.

- Use of architectural style in the assignment.

- Your understanding of the kinds of changes easily supported by the architecture.

In particular, the grade will be broken down as follows (100 points maximum):

- the program: 40 points,

- question 1: 20 points,

- questions 2-5: 10 points each.

6 Further questions

If you have any further questions, feel free to contact any of us via e-mail or during our office hours. Clarifications (if any) will be posted to the class bulletin board.

Code Location: /gs30/usr0/mwang/architectures/assm1/base_1_2

Executable Image: session

Team Members:
- Kent Sarff
- Hung-Ming Wang
- Rob Wojcik
- Rachad Youssef
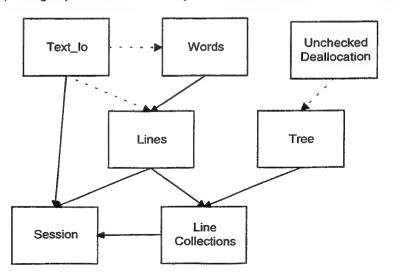
1. Describe the architecture of your system (both the provided parts and the parts you added), explaining how it is an example of object-oriented architecture, and in what ways (if any) it deviates from the basic object-oriented style. For each of the new functionalities required, describe how your system implements it.
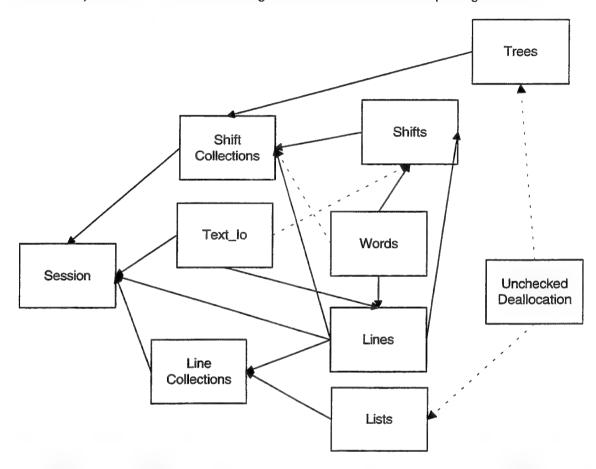
[Note: When Ada package names are used, they are italicized. Example: package *words*.]

The initial system is an example of an object-oriented architectural style for a number of reasons. First, information about the implementation of data types and their operations was hidden from data type users (information hiding). Second, each package is responsible for maintaining the integrity of the ADT's information. Lines, for instance, know about words, but do not know what words are made of and cannot directly manipulate the contents of words. All interfaces to words are through the specifications defined in *word*'s package specification. The design of the initial system does not deviate from this style.

This figure represents the Ada 'withing' structure of the original system. A solid line represents when one package 'withs' another from its specification. A dotted line represents when one package 'withs' another from its body. In general, solid lines show package visibility with respect to external interfaces, and dotted lines show interfaces regarding implementaion details which are usually hidden from other packages. Solid lines to main programs show normal 'with' relationships. A package specification and body combination are shown as one box.

The new system maintains the object-oriented architectural style. New functionality was added by the addition of routines in existing packages and by the addition of new packages which implement new kinds of data. To retain the object-orientedness of the system, rules regarding information hiding and data abstraction were used when designing the new and modified interfaces. To maintain the architectural style, we separated the new desired functionality into two groups. The first group of functionality could be implemented by modifying existing packages. An example of this kind of modification is the additional printing modes (capitalized and abbreviated) for words. This kind of change was localized to the *words* package.

The second kind of modifications required new code. Circular shifts, for example, require more and different internal attributes than lines. Since circular shift was a new kind of data, we created a new package (using the style of the *lines* package), *shifts*, to represent that data. Because a circular shift is partially composed of a line, the *shifts* package body references the *lines* package. Maintaining the architectural style, units that use circular shifts, however, don't know that *shifts* uses lines. We created another new package, *shift_collections*, that hides the implementation of sorted circular shifts.

Regarding the modifications:
  1. Print should output an alphabetical list of all the circular shifts (3 options).

      This functionality required the addition of two new packages and modifications to one package.

The new *shift_collections* package implements the shift collection ADT. It is patterned after the *line_collections* package and provides a number of procedures that add and remove all of a line's non-trivial circular shifts to and from a collection, and prints (3 options) the contents of shift collection. *Shifts* implements the circular shift ADT and is patterned after *lines*.

A shift determines if it is trivial by using a new procedure in the *words* package which returns a boolean stating if a specific word is trivial. This implementation preserves the object-oriented style by hiding the implementation of *is_trivial* in *words*. Only words should know if they are trivial.

The three printing options are implemented in the *words* package as additional print procedures *Print_Abbreviated* and *Print_Capitalized*. The simple print option uses the existing print procedure.

2. Add command *Original*

Printing the lines in the order in which they were added required a modification to the specification and implementation of line collections. The existing package specification did not require changes for this modification. The internal implementation, however, was changed to a linked list by the removal of the tree package (*Tree_Binary_Unbounded_Managed*) and replacing it with the repository linked-list package *Lists*. *Line_Collection*'s package body was modified to instantiate the iterator provided by the linked-list package for printing.

This functionality required modifications in the main routine, *session*, so that a user could select to perform it.

3. Add command *Delete*

This functionality required new code in *shift_collections*, described earlier and modifications in the main routine, *session*, so that a user could select to perform it. *Line_collections* was modified to export an exception if the user tried to delete a line that was not in the line collection. If the line was not in the collection, *Session* trapped the exception and would not attempt to delete any circular shifts possibly associated with the line. This functionality is important as many different original lines could produce identical circular shifts (but not identical sets of circular shifts).

Deleting a shift from a shift collection required a bit of advanced Ada knowledge. If a package T exports a private type to another package using (using a 'with' relationship), and package A instantiates a generic package with type T and the generic package requires the use of the equals "=" operator for type T, package T must explicitly export the equals operator and the operator must be specified as a generic parameter to the generic package.

This situation required the modification of *Lines* to explicitly export an equals operator and the modification of the tree package to import the equals operator.

4. Add command *Count*

This functionality required the addition of a routine to the *line_collection* package which called a routine in the *lists* package. This functionality required modifications in the main routine, *session*, so that a user could select to perform it.

2. For each of the changes you made, explain if the change was of one of the internals of one of the system components (data structure or algorithms) or of the system architecture.

In general, new kinds of data were implemented by changing the system's architecture by the introduction of new components. Examples of this are circular shifts and shift collections. When one of the new requirements regarded the reorganization of data (e.g. it made sense to implement line collections in a list structure), a component was swapped for another. Sometimes, additional functionality required a new method for a kind of data. An example of this change is the addition of the equals operator to the *lines* package. The representation of lines did not change, information hiding and integrity was maintained, but the package was expanded to include a new interface. This could be regarded as an algorithmic change, but it really is only an interface change.

Specific information about each change is identified in the answer to question 1.

3. What changes would you have to make to your system to change the representation of line storage? What other components would be affected?

The answer depends on the type of change. If the assumption that a line is comprised of words is not altered, then changes in line representation would appear in the body of the lines package. Depending upon the scope of the change, additional packages may be added, but those decisions are hidden from units that use the *lines* package.

If the assumption that a line is comprised of words changes, the architecture of the system will change. *Words* are used by the *lines* package, the *shifts* package, and are 'withed' into the package body of *shift_collections*. As a result, if lines are no longer comprised of words, the architecture of the system will change -- in this case, rather dramatically.

4. What changes would you have to make to your system to add the functionality of showing only lines that start with a particular word?

If the change affects only the printing of original lines, it would require an additional interface into the *line_collections* package that would require the 'withing' of the *words* package. The package body implementation of the interface would instantiate an iterator that would visit each item in the collection, and print it using the current routine if its first word matched the one supplied by the user.

If the change would affect the printing of circular shifts, a similar modification would be made to the *shift_collections* package.

5. Does the architecture lend itself to a distributed implementation? If so what changes would have to be made to make it function this way?

Because the data we call line collections and the data we call shift collections are fundamentally disjoint, this architecture does lend itself to distribution. One could imagine an implementation where one machine (or set of machines) provides services regarding line collections and another (set of) machines provides services regarding shift collections. Client programs, running on any number of workstations, would access the services provided by the servers.

The system would require extensive modifications to support this kind of change. There may be may be many clients concurrently reading the system's information, but only one client should be adding or deleting lines or circular shifts at any given time. There are a number of ways to

accomplish this, but the concept of a transaction system that provides ACID properties would do very nicely.

Answering this question brings up many more. The architectural style of the system as we implemented is called object-oriented because it exhibits those properties we call object-oriented. If it is modified in the manner suggested in this answer, it could be called client-server. Is this an architectural style? Is the system now client-server, object-oriented, or both (marketers would *love* to sell an object-oriented, client-server, distributed-repository system...). Can we say that a system has one overall architectural style or do many apply at even the highest level of abstraction?

Assignment 2
KWIC Using a Pipe-Filter Architecture
Due: Wednesday, February 23.

1   Description of the problem

This assignment is to implement an interactive version of the KWIC index system (described in Parnas's *On the Criteria To Be Used in Decomposing Systems into Modules*) in the **Pipe-Filter** paradigm. You will be provided with two implementations of the KWIC system - one in Unix shell commands and one in C. You will be asked to extend these implementations with new functionality.

Both versions of the current system accept input at the command line and produce output to the terminal screen. Both versions implement a pipe and filter system that shifts and sorts the input, and then transforms to upper case letters the first word in each line (look at the first example below).

You will be provided with the source code for these systems, as well as source code for two utility programs, *diverge* and *converge* - one to split an input stream and one to join two input streams.

The source code for the current system will be made available to you by next class period. Watch the class bulletin board for instructions on how and where to obtain the code.

Your assignment is to modify the existing code to support the following changes:

1. Extend the shell script version of the system to produce a KWIC index of the login and user names of all users currently logged on a system. Hint: look at *finger* and *cut* and *tail*.

2. Do the same with the C language implementation.

3. Modify the shell script version of the system to produce a KWIC listing that contains no duplicate entries. Hint: look at *uniq*.

4. Modify the C language version of the system to produce a KWIC index in which the login names of users appear as separate entries from the users' real names. Define a set of "trivial" login names to contain "john","smith", "david" and your own login name. In the final output, only nontrivial login names should appear, and only the first and last name of each real user name should appear. (i.e remove middle initials or middle names). Hint: Use the *diverge* and *converge* programs provided. You might find the C function "nxtarg" usefull for some of

Experience with a Course on Architectures for Software Systems   27-1

the C functions you will have to write, or look into diverge.c to see how parsing
words is easily done.

Here are sample outputs for the solution to each part of the problem:
(The finger part is a real snapshot of some machine, therefore the choice
of names has no deep meaning)

```
% solution1.csh and solution2
BENNETT jcrb John C R
C R Bennett jcrb John
CERIA santi Santiago
DAFNA Talmor tdafna
DAFNA Talmor tdafna
EHT Eric Thayer
ERIC Thayer eht
GALMES pepe Jose M
JCRB John C R Bennett
JOHN C R Bennett jcrb
JOSE M Galmes pepe
M Galmes pepe Jose
PEPE Jose M Galmes
R Bennett jcrb John C
SANTI Santiago Ceria
SANTIAGO Ceria santi
TALMOR tdafna Dafna
TALMOR tdafna Dafna
TDAFNA Dafna Talmor
TDAFNA Dafna Talmor
THAYER eht Eric

% solution3.csh
BENNETT jcrb John C R
C R Bennett jcrb John
CERIA santi Santiago
DAFNA Talmor tdafna
EHT Eric Thayer
ERIC Thayer eht
GALMES pepe Jose M
JCRB John C R Bennett
JOHN C R Bennett jcrb
```

```
JOSE M Galmes pepe
M Galmes pepe Jose
PEPE Jose M Galmes
R Bennett jcrb John C
SANTI Santiago Ceria
SANTIAGO Ceria santi
TALMOR tdafna Dafna
TDAFNA Dafna Talmor
THAYER eht Eric

Remark: TRIVIAL_NAMES = {'tdafna','john','smith','david'}
% solution4
Bennett John
Ceria Santiago
Dafna Talmor
Dafna Talmor
eht
Eric Thayer
Galmes Jose
jcrb
John Bennett
Jose Galmes
pepe
santi
Santiago Ceria
Talmor Dafna
Talmor Dafna
Thayer Eric
```

2   Discussion

On Wednesday, February 16, one teams will briefly present their initial designs for
class critique and discussion. Volunteers for this presentation will be solicited during
the previous class period. Volunteers will be drawn from those groups that did not
present designs for assignment 1.

*This presentation/discussion will not be graded. It is solely for the benefit of you
and your classmates.*

3 Due date and electronic hand-in

The assignment is due by 10:30am on Wednesday, February 23. You should e-mail your solution to the Teaching Assistant by that time. Your solution should consist of

- the names of team members,

- the directory holding the solution.

Your directory should contain 4 text files (besides the c or csh files): "solution1","solution2", "solution3" and "solution4". Each one should a list of the files you use for the solution, with an indication which file is changed or new. All your changes should be well documented within the files.

In addition, there will be a written commentary (due at the beginning of class on Wednesday, February 23) answering the following question:

1. How can the efficiency of the "no duplicates" implementation be changed by using the *sort* and *uniq* filters at different points in the system? (The sorting algorithm has O(n log n) complexity).

The commentary should be your own work: i.e., individuals, not teams for commentary.

4 Grading criteria

Your solutions and commentary will be graded by the following criteria:

- Whether or not the resulting system performs as required.

- Use of architectural style in the assignment.

- Your understanding of the implications of changes made to the system architecture.

In particular, the grade will be broken down as follows (100 points maximum):

- the program: 80 points, and

- question 1: 20 points.

5 Further questions

If you have any further questions, feel free to contact any of us via e-mail or during our office hours. Clarifications (if any) will be posted to the class bulletin board.

1. How can the efficiency of the "no duplicates" implementation be changed by using the *sort* and *uniq* filters at different points in the system? (The sorting algorithm has O(n log n) complexity).

Before discussing efficiency, we must review the requirements of the implementation. These are that the program produce a KWIC index of the login and user names of current users with no duplicates. Any gain in efficiency that violates the requirements can not be considered. The other thing to keep in mind is the precondition to uniq that it operate on a sorted dataset. The combination of these two considerations is that either uniq be the last element in the pipeline, or we must guarantee that no filters after uniq produce multiple entries and that we must guarantee sorted input to uniq.

The architecture that we chose is:

$$\text{finger -f | cut -c1-31 | cshift | sort -f | upcase | uniq} \qquad (1)$$

This solution was chosen because it satisfies the requirements and efficiency is not a concern in our implementation.

Although there are many pipelines containing these filters that will satisfy the requirements, we will consider only one other solution:

$$\text{finger -f | cut -c1-31 | sort -f | uniq | cshift | sort -f | upcase | uniq} \quad (2)$$

Our argument for this selection is that if removing elements from the dataset will improve efficiency, then we should move uniq as far as we can toward the front of the pipeline. However, we want to maximize the possibility of removing duplicate lines. so we cut before the first call to uniq. If there is a big efficiency win in changing the order of the filters it will come from removing duplicate entries before the cshift. For this analysis consider whether or not it will improve efficiency to remove duplicates before cshift. For large datasets, we guess that the answers are yes if there are "lots" or duplicates and no if there are "not many" duplicates. The problem becomes defining "lots" and "not many" and verifying our assumption.

We now must define some variables. Let n be the number of lines output from finger. Let w be the total number of words output from

$$\text{finger -f | cut -c1-31}$$

which is also the number of lines after the cshift filter and let m be the average number of words per line, i.e.

$$w = n * m.$$

Finally let u be the number of unique lines. This is the number of lines that would result from a first call to uniq. We assume that the average number of words per line is the same whether or not duplicates are included. We make one further assumption, if the number of lines is large the total number of original lines and the total number of unique original lines are both much greater than the average number of words per line,

$$n \gg m$$
$$u \gg m.$$

Now consider the order of the computation time of the filters. We are told that the sort is $O(n \log n)$. Because the dataset must be sorted, we assume that uniq is $O(n)$. We also assume that finger, cut and upcase are $O(n)$. cshift is $O(n*m)$ and it also changes the number of lines for later elements in the pipe. When combining the computation times of the filters we will treat them as if they are sequential. For the first solution,

$$\text{finger -f | cut -c1-31 | cshift | sort -f | upcase | uniq} \qquad (3)$$

the computation time is

$$O(\max(n, \; n, \; n*m, \; (n*m)\log(n*m), \; n*m, \; n*m)) \qquad (4)$$
$$= \qquad O((n*m)\log(n*m)) \qquad (5)$$
$$= \qquad O((n*m)\log n + (n*m) \log m) \qquad (6)$$
$$= \qquad O(n*m \; \log n) \qquad (7)$$

For the second solution,

$$\text{finger -f | cut -c1-31 | sort -f | uniq | cshift | sort -f | upcase | uniq} (8)$$

the computation time is

$$O(\max(n,\ n,\ n(\log\ n),\ n,\ m*u,\ (m*u)\log(m*u),\ m*u,\ m*u)) \quad (9)$$
$$=\quad O(\max(n(\log\ n),\ (m*u)\log(m*u))) \quad\quad (10)$$
$$=\quad O(n\ \log\ n) \quad\quad\quad (11)$$

This last follows because $n >= u$ and $n >> m$.

It turns out that all that we are doing in either solution is altering the constant associated with the order of the operation. This analysis has not been much help in defining "lots" and "not many", so we retreat to simpler logic. If u is approximately equal to n then although the order of the operation hasn't changed we are spending twice as much time sorting for no added benefit. If $u << n$ then we do one sort on n elements and then shift only u elements and operate after the shift with $u*m$ elements.

The final summary is that if n is large, the choice of the pipeline architecture depends on whether we expect a large or small percentage of duplicates. If n is small, the simplicity of having fewer filters in the pipeline will outweigh any benefits of removing duplicates earlier in the process.

Assignment 3
KWIC Using an Implicit Invocation Architecture
Due: Monday, March 14.

1   Description of the Problem

This assignment, once again, is to implement an interactive version of the KWIC index system (described in Parnas's *On the Criteria To Be Used in Decomposing Systems into Modules*) in the implicit invocation paradigm. You will be provided with a partial Ada implementation of the system and asked to identify and make the necessary modifications.

The provided system, as in Assignment 1, is simply a line alphabetizer. It interactively inputs a line at a time and upon demand outputs an alphabetized list of the current collection of lines. Unlike the first assignment this version also allows a delete command. Here is a transcript of a sample session:

```
Add, Delete, Print, Quit:
a
> Star Wars
Add, Delete, Print, Quit:
a
> The Empire Strikes Back
Add, Delete, Print, Quit:
a
> Return of the Jedi
Add, Delete, Print, Quit:
p
Return of the Jedi
Star Wars
The Empire Strikes Back
Add, Delete, Print, Quit:
d
> Star Wars
Add, Delete, Print, Quit:
p
Return of the Jedi
The Empire Strikes Back
```

Experience with a Course on Architectures for Software Systems   28-1

Your assignment is to modify the existing code to support the following changes:

1. Rather than simply outputting an alphabetic list of all the lines, the *Print* command should output an alphabetic list of the circular shifts of all the lines. However, shifts (including the nullary shift) which result in a line beginning with a trivial word—*a, an, and, the* and the capitalized versions of these words— should be omitted.

2. On a *Print* command the system should also print a counter of the number of original lines added by the system.

Here is a sample session of the new system:

```
Add, Delete, Print, Quit:
a
> Star Wars
Add, Delete, Print, Quit:
a
> The Empire Strikes Back
Add, Delete, Print, Quit:
a
> Return Of The Jedi
Add, Delete, Print, Quit:
p
-- Number of Original Lines:    3--
Back The Empire Strikes
Empire Strikes Back The
Jedi Return Of The
Of The Jedi Return
Return Of The Jedi
Star Wars
Strikes Back The Empire
Wars Star
Add, Delete, Print, Quit:
d
> Star Wars
Add, Delete, Print, Quit:
p
-- Number of Original Lines:    2--
Back The Empire Strikes
```

```
Empire Strikes Back The
Jedi Return Of The
Of The Jedi Return
Return Of The Jedi
Strikes Back The Empire
```

2   The Current System

The current system is decomposed into the following modules:

- Words

- Lines

- Line_Collections

- Alphabetized_List

- KWIC_Session

In addition, the following additional modules will be used in the final system (they have already been written for you):

- Shifter_1

- Shifter_2

- Trivial_Eater

There is also a file, called event_bindings.ada which contains the bindings from events to methods. **To complete your solution, you should modify this file only, and add one new module.**

You will also need to generate the event manager itself. This is automatically generated from the event description language embedded in the Ada code and in event_bindings.ada. To generate the event manager, type:

```
make_events *.ada
```

This will create two files: event_manager.ada and event_manager.adb. They should be compiled into your system as well.

The format of the event description language is as follows:

- All lines in the event description language are preceeded by the --! symbol. This symbol indicates to Ada that these lines are to be ignored, and to the event description language processor (which is made primarily of awk scripts) that these lines are to be processed. Note that event_bindings.ada contains nothing other than lines in the event description language.

- Each section of the event description language is bracketed by two lines that indicate what package the enclosed declarations are associated with. These lines are:

```
--! for <package_name>
--!  ...
--! end for <package_name>
```

  where <package_name> represents the Ada package name of the associated package. All other declarations go between these two statements (where the ellipsis is).

- To create a new event in the system, include a declare statement of the form:

```
--!   declare <event_name> <args>
```

  where

  - <event_name> represents the name of the event, and
  - <args> represents the data associated with that event (if any). Each argument is of the form:

```
<identifier> : <type>;
```

    where

      * <identifier> is an Ada identifier for the datum, and
      * <type> is the Ada type name of the type of the datum.

All of the event declarations required for this system are included in the specifications of the various packages provided. You should not have to add any on your own.

- Bindings from an event to a method associated with that event can be found in event_bindings.ada. For each binding, the following format is used:

```
--!  when <event_name> => <method_name> <argnames>
```

where

- <event_name> represents the name of the event upon whose announcement the method should be called.
- <method_name> represents the name of the procedure (within the package specified by the for statement) which should be called when the event is announced.
- <argnames> is a list of the identifiers of data associated with the event in a declare statement which are to be passed to the procedures. You do not have to pass every argument, nor do you need to pass them in the same order they are defined. However, every name which appears in <argnames> must have been part of the event declaration.

When a component wishes to announce an event, it calls Announce_Event, signaling the name of the event and any parameters. (All calls to Announce_Event have already been provided in the code. It will help you in your solution to know that this particular implicit invocation system guarantees that whatever activity was caused by the event announcement is complete when the Announce_Event procedure returns, so that there are no pending events in the system once the call returns.

3   Discussion

On Monday March 7, a team will briefly present their initial designs for class critique and discussion. Volunteers for this presentation will be solicited during the previous class period. Note that each team will be responsible for one such presentation over the course of the three assignments.

This presentation/discussion will not be graded. It is solely for the benefit of you and your classmates.

Experience with a Course on Architectures for Software Systems   28-5

4   Due Date and Electronic Hand-In

The assigment is due by 10:30am on Monday March 14. You should e-mail your solution to the Teaching Assistant. Your solution should include:

- the names of your team members,

- a pointer to a directory containing a modified source of event_bindings.ada, the added module, and a running system.

In addition, there will be a written commentary (due at the beginning of class on March 14) answering the following questions:

1. Are implicit systems easier or harder to modify than object-oriented architectures? Why? Describe specific modifications (other than the one which you performed) which would be easier in an implicit invocation system, and other modifications which would be harder.

2. Could the system specified be implemented using a dataflow architecture? If so, how? If not, why not?

3. Explain how your implementation differs from the one proposed in the paper by Garlan, Kaiser, and Notkin for handling trivial line removal. Would that have been a better approach? If so, why? If not, why not?

4. The implicit invocation system provided by make_events assures that all events which are caused by a single Announce_Event, whether directly or indirectly, are all complete and all methods called before the Announce_Event call returns. Identify any differences in your solution which would have been caused if the system delivered the events in arbitrary order, and did not guarantee their delivery prior to returning from an announcement.

5. Does your system handle line deletions properly? Defend.

The commentary should be your own work; i.e., individuals, not teams for commentary.

5   Grading Criteria

Your solutions and commentary will be graded by the following criteria:

- Whether or not the resulting system performs as required.

- Use of architectural style in the assignment.

- Your understanding of the kinds of changes easily supported by the architecture.

In particular, the grade will be broken down as follows (100 points maximum):

- the program: 40 points,

- questions 1-5: 12 points each.
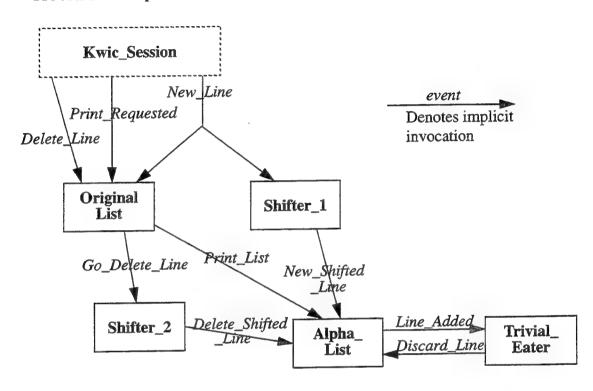
6   Further Questions

If you have any further questions, feel free to contact any of us via e-mail or during our office hours. Clarifications (if any) will be posted to the class mailing list.

Project 3 - Comments

François Truchon

FIGURE 1. Our implementation

1.0 Implicit systems vs. object-oriented architectures

The implicit invocation mechanism has the advantage that it allows someone to put a system together by simply connecting the various components together much like a shell script does in pipe and filter systems. Since the system can be modified by editing a single file (the event bindings file), modifications are eased considerably. This is in comparison to a typical object-oriented architecture where changing the calling structure in the system usually entails making modifications to many modules. Object-oriented systems can become difficult to modify because one has to look through the code to find the architectural connections. With this implicit invocation system, you can simply look at the event bindings.

One of the difficulties with implicit invocation is exemplified by the delete operation. Here, we want to delete shifts only when we are certain that the line to be deleted had already been added to the system. In effect, we want to impose a condition on delivery of event **Delete_Line**. Although the solution to this particular problem was to simply create a

chain of events, had we wanted to add multiple conditions this could have become very difficult. This is a problem that a typical object-oriented could have dealt with fairly easily.
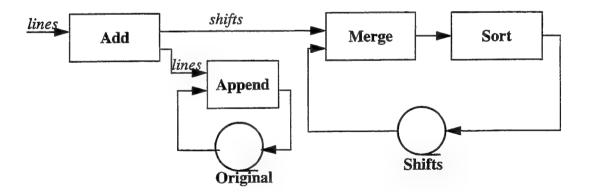
A modification that would be easily done in an implicit invocation system would be to only add unique lines to the system (i.e. not allow duplicates). This could be accomplished by binding event **New_Line** to method Original_List.add_uniq() which generates event **New_List_X**, itself bound to Shifter_1.generate_Shifts. The only file to be modified is the event bindings file (assuming that the methods and events used are already implemented).

A modification that would be hard to implement using implicit invocation would be not to allow the user to add lines that only contain trivial words. For instance, adding "The a the" would be rejected or ignored. We could probably conceive of some sequence of events involving a Shifter, a Trivial_Eater and Original_List but this would be fairly messy. In an object-oriented system, this would likely be an easy change.

2.0 Dataflow architecture

I believe that a dataflow architecture would be completely inappropriate for this system given its interactive nature. The purpose of the system is to allow a user to interactively edit the kwic database and this does not lend itself well to a pipe and filter architecture or a batch sequential architecture.

Of course if we really wanted to, we could devise batch operations to perform adds, deletes and prints operating on two tapes: Original and Shifts. A possible dataflow implementation of add would be:

The delete and print could be defined in a similar fashion.

3.0 Trivial line removal

In our implementation, a new line shift is first added to Alphabetized_List, then event **Line_Added** is generated which triggers Trivial_Eater to examine the shift. If the new shift

starts by a trivial word, it is deleted. The problem is that the trivial shift is inserted only to be deleted immediately. This differs from the implementation proposed in the paper to generate an event *before* the shift is inserted and abort the insert operation if the shift is found to be trivial. So here we could generated event **Adding_Line** which would trigger Trivial_Eater.

The problem resides in aborting the operation. We cannot simply return **Abort** (for instance) to inform the Alphabetized_List that it must abort since the current event mechanism doesn't support return values. We could, on the other hand, use the Ada exception handling mechanism to abort the insert operation. This would be a somewhat unorthodox use of exceptions but would work as long as exceptions are declared and caught properly. Another alternative would be to abort the operation by setting a flag in Alphabetized_List that indicates to the insert operation that it should abort. This last alternative, although not very elegant would probably be the simplest.

Given that adding and deleting a line from the b-tree can be expensive, aborting the insert operation would probably have been a better approach and not that difficult to implement. It would be important to make sure that the abort mechanism (or better some form of event return value) is implemented neatly and consistently so that the use of abort not be just a hack for efficiency. Indeed, the reason for introducing implicit invocation in the first place was to easy system maintenance, not to make it more difficult by introducing hacks for efficiency.

4.0 Event Delivery

In the current implementation, adding and deleting the same line results in the following cascade of events (notation is a mix of CSP and regular expressions):

New_Line -> (New_Shifted_Line -> Line_Added -> Discard_Line^{0,1})*

Delete_Line -> Go_Delete_Line -> Delete_Shifted_Line*

If the order of delivery was not guaranteed anymore, we could end up with an invalid interleaving of these sequences of event. For instance:

New_Line -> Delete_Line -> Delete_Shifted_Line* -> (New_Shifted_Line -> Line_Added -> Discard_Line^{0,1})*

The problem here is that the system tries to delete the line shifts before they are even added, the result being that the line shifts remain in the system.

To prevent this, we could implement a locking mechanism that serializes the critical operations. For this problem, coarse-grained locking on user events would likely be sufficient. For instance, the Kwic_Session module could interact with an Ada task with entries for each basic operation: add, delete and print. This task would not rendez-vous on any of the operations until the previous one has completed.

5.0 Line deletion

One of the problems posed by line deletion is in deleting only the shifts that were created from the same line and not other shifts.

For instance, if the system contains only one original line:

the king is dead

and the user tries to delete:

dead the king is

the system must not delete the shift the originated from *"the king is dead"*. We ensure that this is the case by deleting the line from Original_List, which announces event **Go_Delete_Line** only if the line was actually present in the list. In effect, we are using Original_List to check that a line is valid.

Our system would not have worked properly had we bound event **Delete_List** directly to Shifter_2.

Also, a problem could occur if Original_List deleted all line duplicates while Alphabetized_List only deleted one line. We would end up with shifts that could not be deleted. But this is not the case.

Assignment 4
Formal Models: Event Systems
Due: April 6

1 Description of the Problem

This assignment is intended to help you develop some experience in manipulating a formal model of a software architecture. In this case you will be using the formal model of event systems presented in class. Following the pattern of specialization in [GN91] you are to formally characterize as event systems the two architectural idioms described below. You may wish to consult the references [Spi89a, Spi89b, PST91] for additional help with the Z notation.

2 Blackboard Systems

Drawing on Nii's description [Nii86a, Nii86b] describe a blackboard system as a formal specialization of *EventSystem*. You may find it helpful to make the following simplifying assumptions:

- There are two kinds of components in a blackboard system: *BBdata* and *ksources*.

- The *BBdata* in the blackboard system are partitioned into a collection of *layers*.

- Each *ksource* is associated with a some set of these layers.

- Each *ksource* has a method *UpdateBB*, which allows it to update the blackboard when it is invoked.

- When the data in a blackboard changes, for each layer that is changed the system announces the *ChangedLayer* event to each of the knowledge sources that are associated with that layer.

You need not say anything about the run time mechanisms involved in carrying out the updates. In particular, you don't have to say how the knowledge sources update the blackboard, or how new data is added to the blackboard.

3   Spreadsheet Systems

Formally characterize a spreadsheet system as an event system. For the purposes of this assignment you can consider a spreadsheet to be an $N \times M$ matrix. Some of the entries in this matrix will have a *VALUE*. Some of the entries will also have an associated *EQUATION* that describes the value of that entry as a function over other entries in the spreadsheet. When spreadsheet entries are changed the equations that depend on those entries are implicitly reevaluated. As with the blackboard, you need not formalize the run time mechanism of a spreadsheet.

You might find the following definitions to be a useful starting point:

$[VALUE, EQUATION]$

$Pos == \mathbb{N} \times \mathbb{N}$

$$
\begin{array}{|l}
Params : EQUATION \longrightarrow \text{seq } Pos \\
Eval : (EQUATION \times \text{seq } VALUE) \nrightarrow VALUE \\
\hline
\forall\, e : EQUATION;\ vs : \text{seq } VALUE\ \bullet \\
\qquad (e, vs) \in \text{dom } Eval \Leftrightarrow \#vs = \#(Params\ e)
\end{array}
$$

In other words, we take *VALUE* and *EQUATION* to be primitive types, and a matrix position, *Pos* to be a pair of natural numbers. We assume (axiomatically) that we can determine for each equation what its parameters are and also how to evaluate it for actual values. (The invariant guarantees that number of formal parameters must match the number of actual parameters.)

With this as a basis you can then define a spreadsheet along the following lines:

$$
\begin{array}{|l}
\underline{SpreadSheet}\rule{6cm}{0.4pt} \\
EventSystem \\
height, width : \mathbb{N} \\
boxes : Pos \rightarrowtail\!\!\!\rightarrow Component \\
eqns : Pos \nrightarrow EQUATION \\
vals : Pos \nrightarrow VALUE \\
\cdots \\
\hline
\cdots
\end{array}
$$

The symbol $\rightarrowtail\!\!\!\rightarrow$ indicates that each position is associated with a unique component.

You may assume that each Component in a spreadsheet (associated with a box via *boxes*) can update its value using the method *Update* whenever it gets the *Reevaluate*

event. Your task is to add any appropriate additional state and the state invariants. In particular, the state invariant should explain how EM is determined by the other parts of the spreadsheet.

4 What to Hand In

You should hand in:

- A description of the two formal models outlined above. Ideally this should be formatted and checked using Fuzz, but it need not be. As with all Z documents, the formalism should be accompanied by enough prose to explain what is going on. You may work in groups to produce this document.

- As individuals you should also turn in commentary addressing the following questions:

  1. What important aspects of the modeled architectures are (intentionally) left out of the model.

  2. One might imagine that an interesting property of a blackboard system would be whether the knowledge sources interfere with each other. For the blackboard system, do you think it would be possible to model some notion of "non-interference?" (You need not model it, but you should explain why or why not you answered the question in the way you did.)

  3. For the spreadsheet system, is the *Circular* property defined in the events paper a relevant concept? Why or why not?

  4. For both the blackboard and spreadsheet models, explain *briefly* which of the other formal event models described in the paper is most similar.

We will make a copy of the Z description for the event system described in [GN91] available to you.

5 Grading Criteria

Your solutions and commentary will be graded by the following criteria:

- Whether or not you are able to model the requested specializations.

- Your ability to understand and explain the formalisms in the accompanying prose.

6 Further Questions

As usual, if you have any further questions, feel free to contact any of us via e-mail or during our office hours. Clarifications (if any) will be posted to the class mailing list.

References

[GN91]   David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44. Springer-Verlag, LNCS 551, October 1991.

[Nii86a] H. Penny Nii. Blackboard systems part 1: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(3):38–53, Summer 1986.

[Nii86b] H. Penny Nii. Blackboard systems part 2: Blackboard application systems and a knowledge engineering perspective. *AI Magazine*, 7(4):82–107, August 1986.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[Spi89a] J. M. Spivey. *The Z Notation: A Reference Manual*, Prentice Hall, 1989.

[Spi89b] J. M. Spivey. An Introduction to Z and Formal Specification. *Software Engineering Journal*, 1(4), January 1989.

Architectures of Software Systems, 15-775
Assignment 4
Formal Models: Event Systems

Kent Sarff   Hung-Ming Wang   Rob Wojcik   Rachad Youssef

June 11, 1994

1   Introduction

This is a *fuzz*-checked document, which can be found via the following path,

  /gs30/usr0/mwang/architectures/assm4/assm4.tex.

The second section establishes our basic event model, and was stolen from [GN91] without any change. The third section describes our specialization of the basic model towards a blackboard system. The fourth section describes our specialization of the model towards a spreadsheet system. The final section was motivated by answering the write-up question 3. We decided to model the anti-circular property explicitly to show our understanding.

In both exercises, the dynamic run-time model is intentionally left out. We only address the static associations between events and methods in the *EM* relation. *EM* relation is just like a bookkeeper. *EM* could be used by a run-time model to implement various invocation policies. We do not, however, say *how* the system behaves.

2   The Basic Model

We begin by assuming there exist sets of events, methods, and component names, which, for the time being, we will simply treat as primitive types.

  $[EVENT, METHOD, CNAME]$

A component is modelled as an entity that has a name and an interface consisting of a set of *methods* and a set of *events*.

```
┌─ Component ─────────────────────────────────────────────
│  name : CNAME
│  methods : P METHOD
│  events : P EVENT
└─────────────────────────────────────────────────────────
```

A particular event (or method) is identified by a pair consisting of the name of a component and the event (or method) itself. In this way we can talk about the same event or method appearing in different components. We use the type abbreviations *Event* and *Method* to refer to these pairs (respectively).

$$Event == CNAME \times EVENT$$
$$Method == CNAME \times METHOD$$

For convenience we define the functions *Events* and *Methods*, which extract the set of components and methods from a collection of components.

$$Events : \mathbb{P}\ Component \longrightarrow \mathbb{P}\ Event$$
$$Methods : \mathbb{P}\ Component \longrightarrow \mathbb{P}\ Method$$

$$Events\ cs = \{c : cs;\ e : EVENT \mid e \in c.events \bullet (c.name, e)\}$$
$$Methods\ cs = \{c : cs;\ m : METHOD \mid m \in c.methods \bullet (c.name, m)\}$$

An event system, *EventSystem*, consists of a set of components and an event manager. The event manager, *EM*, is a binary relation associating events and methods that should be invoked when that event is announced. Thus, as we will see later, when an event e is announced, all methods related to it by *EM* are invoked in the corresponding components.

___ EventSystem _____
$$components : \mathbb{P}\ Component$$
$$EM : Event \longleftrightarrow Method$$

$$\forall c_1, c_2 : components \bullet (c_1.name = c_2.name) \Leftrightarrow (c_1 = c_2)$$
$$\mathrm{dom}\ EM \subseteq Events\ components$$
$$\mathrm{ran}\ EM \subseteq Methods\ components$$

The state invariant of *EventSystem* asserts that the components in the system have unique names, and that the event manager contains only events and methods that actually exist in the system.

3 Blackboard Systems

Each knowledge source has a method *UpdateBB*, which allows it to update the blackboard when it is invoked. A *ChangedLayer* event can be announced to trigger knowledge sources.

$$UpdateBB : METHOD$$
$$ChangedLayer : EVENT$$

In our model, there are two kinds of components in a blackboard system. *ksources* is a set of knowledge sources, each of which is a component. *BBdata* is the blackboard structure.

In a pure blackboard model, there is only one blackboard structure. We assume, however, that the blackboard structure is further partitioned into a collection of layers, each of which is a component. A knoweldge source can show its interest in a particular layer by registering to *layer_mapping*. *layer_mapping* is a function which relates each layer in *BBdata* to a set of knowledge sources which are interested in that layer.

```
┌─ Blackboard ─────────────────────────────────────────────────────────
│  EventSystem
│  ksources : ℙ Component
│  BBdata : ℙ Component
│  layer_mapping : Component ⇸ ℙ Component
├──────────────────────────────────────────────────────────────────────
│  BBdata ∪ ksources = components
│  BBdata ∩ ksources = ∅
│  dom layer_mapping = BBdata
│  ∀ ks_assoc_w_layer : ran layer_mapping • ks_assoc_w_layer ⊆ ksources
│  ∀ k : ksources • UpdateBB ∈ k.methods
│  ∀ l : BBdata • ChangedLayer ∈ l.events
│  EM = { l : BBdata; k : ksources
│              | l ∈ dom layer_mapping ∧ k ∈ layer_mapping (l)
│                  • ((l.name, ChangedLayer), (k.name, UpdateBB)) }
└──────────────────────────────────────────────────────────────────────
```

The first two predicates indicate that *BBdata* and *ksources* are a partition of all components in the system. The third predicate says that the mapping exists for each layer of the blackboard. The fourth predicate says that the mapping maps a layer to only knowledge sources. The next two predicates indicate that each knowledge source has an *UpdateBB* method and each layer should be associated with a *ChangedLayer* event.

With the above definition, the *EM* relation can be precisely determined. *EM* simply pairs a *ChangedLayer* event of a layer to the *UpdateBB* methods of those knowledge sources which already show their interests in that layer in the mapping, *layer_mapping*.

4 Spreadsheet Systems

In a spreadsheet system, each cell has a value of type *VALUE*, or an equation of type *EQUATION*. Each cell is identified by its position, of type *Pos*.

[*VALUE, EQUATION*]
$Pos == \mathbb{N} \times \mathbb{N}$

We assume axiomatically that we can determine for each equation what its parameters are as a sequence of positions.

$Params : EQUATION \rightarrow \text{seq } Pos$

Each cell can update its value using the *Update* method whenever it gets the *Reevaluate* event.

> *Update* : *METHOD*
> *Reevaluate* : *EVENT*

The spreadsheet is comprised of a *height* × *width* matrix of cells. Each cell is modelled as a component. *boxes* can be used to identify a cell by giving it a position. *eqns* is a partial function relating a position to an equation. Similarly, *vals* is a partial function relating a position to a value.

$$
\begin{array}{l}
\_Spreadsheet _____ \\
EventSystem \\
height, width : \mathbb{N} \\
boxes : Pos \rightarrowtail Component \\
eqns : Pos \nrightarrow EQUATION \\
vals : Pos \nrightarrow VALUE \\
\rule{9cm}{0.4pt} \\
components = \operatorname{ran} boxes \\
\# boxes = height * width \\
\operatorname{dom} eqns \cup \operatorname{dom} vals = \operatorname{dom} boxes \\
\operatorname{dom} eqns \cap \operatorname{dom} vals = \varnothing \\
\forall c : \operatorname{ran} boxes \bullet Update \in c.methods \land Reevaluate \in c.events \\
\forall c_1, c_2 : components \\
\qquad \bullet ((c_1.name, Reevaluate), (c_2.name, Update)) \in EM \\
\qquad\qquad \Leftrightarrow (\exists p_1 : \operatorname{dom} boxes; \; p_2 : \operatorname{dom} eqns \mid boxes\, p_1 = c_1 \land boxes\, p_2 = c_2 \\
\qquad\qquad\qquad \bullet p_1 \in \operatorname{ran}(Params(eqns\, p_2)))
\end{array}
$$

The first predicate indicates that *boxes* records all cells in the spreadsheet as components. The second predicate indicates that the *boxes* records all *height* × *width* cells in the system. The next two predicates indicate that each cell has either a value or an equation but not both. The next predicate indicates that each cell has an *Update* method and is associated with a *Reevaluate* event. (Note that, instead of saying each "equation" cell has an *Update* method, we say each cell has an *Update* method because the *Update* method could possibly be used for explicit invocation to a "value" cell. For example, when a user enters a value to a cell, it would be necessary to invoke the *Update* method of that cell to update its value.)

With the above definition, the *EM* relation can be precisely determined. When a cell is reevaluated, all cells having an equation which needs a parameter of that cell should be updated. *EM* simply pairs a *Reevaluate* event of a cell to the *Update* methods of those cells which has an equation, and the parameter list of the equation includes that reevaluated cell.

5 Spreadsheet Without Circularity

We do this additional exercise to show we understand the *Circular* property. This is very straightforward. We add a relation, *dependents*, which records all associations between an equation cell and all its parameter cells.

SpreadsheetWithoutCircularity _____

Spreadsheet
$dependents : Pos \leftrightarrow Pos$

$dependents = \{\, p : \text{dom } eqns;\ q : \text{dom } boxes$
$\qquad\qquad\qquad | \ q \in \text{ran}\,(Params\,(eqns\,p)) \bullet p \mapsto q \,\}$
$\forall\, p : \text{dom } boxes \bullet (\,p \mapsto p\,) \notin dependents^{+}$

    In order to guard against circular reference, we just add one more state invariant. The last predicate says that a cell cannot be in the transitive closure of *dependents*. This ensures that a cell cannot eventually make a reference to itself.

1. What important aspects of the modeled architectures are intentionally left out of the model.

(1) In both exercises, the dynamic run-time model is intentionally left out. We only address the static associations between events/methods in the *EM* relation. **We do not say how it behaves.** We do not say who will announce an event and how the system choose an event and then invoke the methods associated with it. There are many possible decisions which can be made for this run-time model: such as the order in which the methods are invoked, whether methods can be invoked concurrently, whether methods can change the set of components in the system, how new events are announced as a side effect of method invocation, etc.

(2) In the blackboard exercise, dataflows are not modelled. We only say that knowledge sources can respond to data changes in the blackboard structure which they are interested in. How they actually get the data is left out. In addition, the individual data items within one layer are not modelled.

(3) In the spreadsheet exercise, how the evaluation of equations is performed is left out. We only address the related cells necessary to participate in the evaluation. But the evaluation process can be done in various ways. In addition, probably some mechanism is needed to guard against circular evaluation (more information in question 3).

2. One might imagine that an interesting property of a blackboard system would be whether the knowledge sources interfere with each other. For the blackboard system, do you think it would be possible to model some notion of "non-interference?"

It is likely that various knowledge sources will interfere with each other. For example, suppose there are 5 knowledge sources interested in a particular layer in the blackboard. When data in that layer is changed, all 5 knowledge sources will be triggered by the *ChangedLayer* event. The problems happen, however, when the 5 knowledge sources want to access the data in that layer simultaneously. It may have **exclusive data access problems** like those often discussed for operating systems.

As mentioned in question 1. We can devise a run-time model to solve this problem. Alternatively, we can augment our current model by imposing an order on the sequence of invocations to knowledge sources. One possible way to model it is,
    Instead of defining *layer_mapping: Component* $\rightarrow$ P *Component*,
    we may define *layer_mapping: Component* $\rightarrow$ seq *Component*.
This will impose an order. Then we need to model a control component to address the

invocation policy about selecting one knowledge source for execution one at a time. This essentially prioritizes the execution of knowledge sources.

Another way to model this is to add some mechanism to control the access of each layer in the blackboard. Only one knowledge source can access the data at any time. This model separates the concern of data access from the implicit invocation of an event system. I prefer modelling in this way.

3. For the spreadsheet system, is the *Circular* property defined in the events paper a relevant concept? Why or why not?

Actually this is a very critical property. Our spreadsheet model, however, is very simple and is **not circularity-free**. It is likely that a chain of implicit invocations that starts at one cell and returns to that cell. This causes a recursive evaluation and will never terminate. In the simplest case, it allows an equation of a cell to refer to itself. For example, a circular reference occurs when a cell of E7 has an equation which needs a parameter of cell E7 itself. In a more complicated case, an equation will possible refer to another cell which in turn refers back to the original cell. This will also result in a circular reference. In essence, an **anti-transitive closure property of references** should hold in order to assure that there are no circularities. Our model does not handle this danger but can be modified to remove this situation. We can define a relation, say, *dependents*, to record all evaluation dependencies between pairs of cells, and does not allow a cell to have an evaluation dependency on itself (some constraint like $(c, c) \notin dependents^+$, we have also modelled this as an additional exercise in our document).

4. For both the blackboard and spreadsheet models, explain briefly which of the other formal event models described in the paper is most similar.

Gandalf is very similar to our blackboard model. Gandalf has two kinds of components: abstract syntax trees (ASTs) and daemons. The user creates a program by incrementally building an abstract syntax tree. As nodes are added to the tree, daemons associated with those nodes are activated to do type checking, provide incremental code generation, etc. Thus ASTs correspond to the layers, and daemons correspond to the knowledge sources. Only nodes in an AST can announce events (only layers can announce events). Daemons handle events from AST nodes (knowledge sources handle events from layers). This analogy is very strong. However, the number of daemons associated with each node is limited in Gandalf but not limited in our blackboard model.

ST80 is similar to our spreadsheet model. In ST80, the *update* method corresponds to our *Update* method in spreadsheet. The *change* event corresponds to our *Reevaluate* event in spreadsheet. The *dependents* relation in ST80 is similar to our *Params* function in spreadsheet. The component dependency in ST80 is just like the evaluation dependency of parameters of cells in spreadsheet. *EM* relation records each of these dependency relationships in both models. So we make this conclusion.

Experience with a Course on Architectures for Software Systems   29-11

Course Project

Project Information

The course project will give you an opportunity to examine and describe the architecture of a real system, the Lunar Rover Demonstration System (LRDS). LRDS is one of the development tasks of the Autonomous Planetary Exploration (APEX) project of the Robotics Institute. The Rover System is being built by the CMU Robotics Institute with the software development assistance of students in the Masters of Software Engineering program.

The Rover System will autonomously explore remote environments (e.g., the Moon). The Rover System will be used for tasks including mapping of an environment, analysis of rocks or soil specimens and imagery provided for entertainment purposes.

The project will focus on two aspects of the of the Rover System:

1. LRDS: This is the overall system to control the Rover and integrate the various functions that it will perform.

2. The Navigation System: This is the subsystem of the LRDS that supports the mission of the Rover in three major operational modes:

a. supervised teleoperation: a user directs the motion of the Rover through the system's user interface.

b. autonomous motion: a user provides a mission to be accomplished, and the Navigation System is responsible for planning the motion of the Rover.

c. teleoperation: an special operator directs the motion of the Rover; it differs from supervised teleoperation in that the operator might have to override some of the safety constraints that would otherwise be active in supervised teleoperation mode.

Group teams have been arranged so that relevant domain expertise of the MSE students in the class are spread evenly among the teams. MSE team members have access to the detailed specifications and design documentation for the LRDS and Navigation System.

The Assignment

Your course project will focus upon the following :

Architectural proposals

1. Examine the software requirement specifications for the Rover System. Based on what you now know about architectural level design, propose two or more software architectural designs for the Rover System to meet its requirements. Discuss the significant features of those proposed architectures and explain how they to address the system requirements. In your discussion pay special attention to the tradeoffs in design that are made in a given architecture: for example, some architectural features may support certain requirements at the expense of others.

2. As above for the Navigation System.

Critique/Analysis

1. How do your architectural designs for the Navigation System fit into the overall architectural design of the LRDS?

2. For both the LRDS and the Navigation System, think of three new requirements not included in the current specification, but which might be incorporated in a future version of the system. Analyze which of the new requirements can be easily accommodated by the architectural designs. Be specific: for example, what components and connectors would be impacted by the new requirements?

3. In view of the above analysis, which (if either) of your architectures is better regarding their ability to handle the anticipated changes?

Project Guidelines

Teams:

As noted above, the teams are made up of people with familiarity with the studio project, and people without previous knowledge of the project. The project is a team effort with no individual write-ups so it is in your interest to find a way to cooperate with your team members and enable everyone to contribute. As we see it, the people familiar with the project can help the team to understand the project, while people who see the project for the first time are in a better position to be able to suggest new architectures, and criticize the existing ones. However, we expect you to find your own way to use skills of project members wisely and organize responsibilities fairly.

Architecture Vocabulary and General Hints:

You should use the vocabulary of the course to characterize the architectures when appropriate. However, you do not need to consider architectural styles that are clearly irrelevant, nor do you need to force your characterizations to conform to any of the ``pure" architectural styles introduced in the course.

Tasks, Dates, and Grading Policy

Preliminary presentation:

To help you make progress and to give you early feedback we would like your team to develop a 15-minute presentation of your preliminary results. The dates for these presentations are April 4 and 6. Two groups will present in each class. This need not be a polished analysis, but it should contain enough substance for us to comment on whether you seem to be on the right track. Be sure to leave time for questions on your presentation. This analysis and presentation will not be graded and will not affect your final grade on the project.

Final write-up:

The final write-up is due at the beginning of class, April 25. The total number of pages for this write-up should be on the order of 20 pages, but should not exceed 25 pages.

Final presentation:

During the classes of April 25 and 27 each group will present its results in a half-hour presentation.

Grading: Your final grade will be based on the final writeup and the final presentation.

Architectures for the Lunar Rover Demonstrator and Navigation Systems

Kent Sarff
Hung-Ming Wang
Rob Wojcik
Rachad Youssef

Final Project
Architectures for Software Systems

April 22, 1994

Abstract: This document describes two candidate software architectures for the Lunar Rover Demonstrator System, two candidate architectures for the Navigation subsystem, decision criteria for selecting a combination of the archtiectures, and the impacts of a number of proposed requirements changes.

Table of Contents

List of Figures

List of Tables

1   Introduction

This document proposes and evaluates software architectures for the Lunar Rover Demonstrator System. In particular, software architectures are presented for the main rover system and the navigation subsystem. All architectures are based on the requirements described in the Rover System Specification [CMU/MSE-TALUS-ROVER-SS, Version 1.0] and the Navigation Software Requirements Specification [CMU/MSE-TALUS-NAV-SRS, Version 2.0].

1.1   Document Overview

Section 1 provides a document overview, a summary of which architectures we recommend, and our assumptions regarding the main rover system and the navigation subsystem.

Section 2 provides a detailed description of the architectures proposed for the main rover system and the navigation subsystem. In all, four architectures are presented; two are for the main rover system: a repository-based architecture and an event-based architecture; two are for the navigation subsystem: a layered architecture and a blackboard architecture. Each candidate architecture is presented in its own section as follows:

- Introduction:

    provides an textual overview and a diagram for the candidate architecture

- Components:

    provides a detailed description of each component in an architecture

- Connectors:

    provides a description of how components interact with each other and external processes

- Operational Scenario:

    provides a description of typical situations the system would be expected to support and how the architecture is constructed to handle them

Section 3 describes the criteria used to choose between the candidate architectures. In particular, a design space and design rules are presented which provide a mapping between desired system characteristics and features of the candidate architectures. In addition, we present justifications why one architecture is better than another for handling individual system requirements. Also the characteristics we felt most important to the main rover system and navigation subsystem are presented along with our final choice of architectures. Finally, this section describes how both rover architectures can be combined with the navigation architectures.

Section 4 describes a few possible requirements changes for the main rover system and the navigation subsystem and how the proposed architectures would support those changes; three potential changes are described for the main rover system along with descriptions of how

those changes would be supported by the event-based architecture and the repository architecture; four potential changes are described for the navigation subsystem and how those changes would be supported by the layered architecture and blackboard architecture.

1.2   Summary of Recommendations

This subsection provides an executive overview of our recommendations and conclusions. Based upon our evaluation of the four architectures presented in this document (please see section 3 for detailed justifications), we make the following recommendations:

1. An event-based architecture should be used for the main rover system. This decision was based on the following criteria:

    * due to the nature of a robotic application, the architecture must support asynchronous processing.
    * because functional and operational requirements are likely to change over time, the architecture must support system evolution including adding new components and reconfiguring existing components.
    * the main rover system will be implemented in a distributed processing environment which may allow multiple processes and CPUs.

2. A blackboard architecture should be used for the navigation subsystem. This decision was based on the following criteria:

    * the planning components will require heuristic knowledge and reasoning as opposed to precise algorithms and deterministic scheduling to perform planning tasks.
    * because the processing requirements for planning, communication, and perception are likely to change as the system evolves, the architecture must support adding new processing approaches.
    * because information about the problem domain is incomplete and necessarily acquired incrementally, the system should support incremental development and refinement.

1.3   Notes

1. We assume the Navigation subsystem provides only planning and perception. The Navigation system may be tasked to obtain detailed maps of terrain within its sensor range. The Navigation system does not provide reflexive behavior. The Navigation system "owns" the black & white camera interface.

2. Different levels of detail are presented for the two rover architectures and for the two navigation architectures. This was done intentionally as the decision to choose specific architectures became clear. It become important to describe our chosen architectures at a finer level of detail.

2 Candidate Architectures

2.1 Rover System 1 - Repository Architecture

2.1.1 Introduction

Our first architectural proposal of the rover system is originated from a repository view. This view emphasizes a coherent state of the system. The motivation of this proposal is simple: the rover can be imagined as a human person, and internally there must be an overall system state at any time. By grouping all state information together, we can easily ensure that the system state is consistent. The architecture is depicted in Figure 2.1.

All information about the current state of the rover is kept in a common repository, for example, the current pose, the maximum allowable speed, the terrain map database, the tilts, the power conditions, the various motion commands, etc. The information is centrally kept and likely will be modified and retrieved by many components.

Each component can be considered a domain expert, which is responsible for performing different tasks. Each component has its special knowledge to handle specific jobs. Each component also has its own secrets with respect to implementation. Please see the next subsection for more explanation.

Each component communicates with each other through the common repository. They do not communicate directly. Each component may be interested in several data items stored in the common repository. Each component can make contribution to the overall rover system. For example, Map Manager can improve the resolution of the environment "seen" by the rover, Reflexive Behavior can prevent the rover from getting into catastrophic situations, Navigation system can suggest the most valuable path to traverse, etc. Each component makes its contribution by retrieving data in the central repository, applying its knowledge to process the data, and modifying data in the central repository.
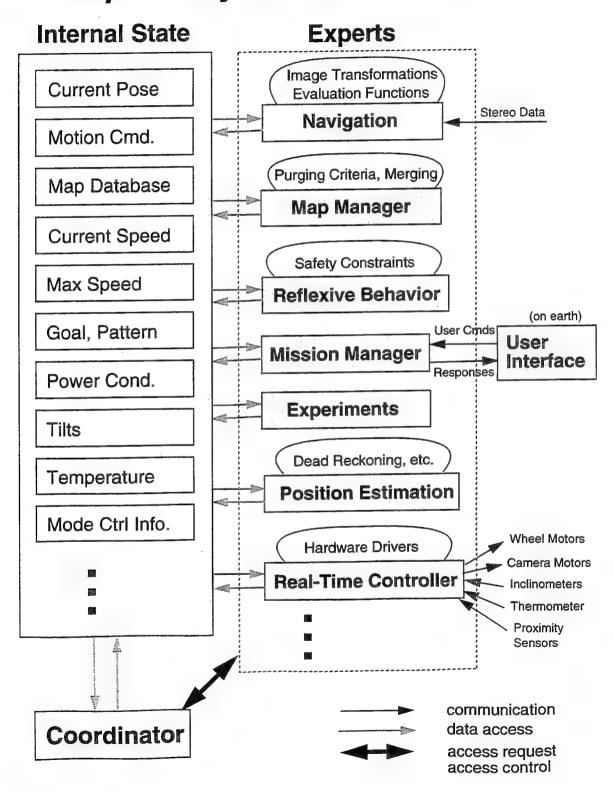
In short, all components in the system (except for User Interface) cooperate with each other implicitly through the common state information, and use the central repository as the media for communication and coordination purposes.

Please note that we named this architecture as a repository, not a blackboard. A blackboard architecture is a specialized form of a repository. We adopted this more general view because:

1. We don't need to perform opportunistic reasoning at this level. Actually it is dangerous to perform opportunistic reasoning in the event of an emergency; we want to say "stop" immediately.
2. The data in the repository are not organized in any hierarchical way.
3. Each component is not triggered by data updates as in a blackboard. Each component has its own thread and operates in parallel. Each component is an active agent.

Figure 2-1 Rover System Repository Architecture

Repository Architecture

Internal State

- Current Pose
- Motion Cmd.
- Map Database
- Current Speed
- Max Speed
- Goal, Pattern
- Power Cond.
- Tilts
- Temperature
- Mode Ctrl Info.

Experts

- Image Transformations / Evaluation Functions — **Navigation** — Stereo Data
- Purging Criteria, Merging — **Map Manager**
- Safety Constraints — **Reflexive Behavior**
- **Mission Manager** — User Cmds / Responses — **User Interface** (on earth)
- **Experiments**
- Dead Reckoning, etc. — **Position Estimation**
- Hardware Drivers — **Real-Time Controller** — Wheel Motors, Camera Motors, Inclinometers, Thermometer, Proximity Sensors

Coordinator

- → communication
- ⇒ data access
- ⟷ access request / access control

2.1.2 Components

This subsection includes a general description of each component depicted in the above diagram. Note that this is only a general description, and a complete list of design specifications should be obtained after further analysis.

1. Central Repository:

   - It keeps all information about the current state of the rover. The information is shared by the various components in the system.

   - The data items depicted in the diagram are not a complete list.

   - It can be considered a cluster of memory, which does not know the identity of the components in the system and is passive in nature.

   - Secret: To ease the changes of various data representations, we can use the abstract data types to maintain each class of data.

2. Navigation:

   - Navigation system provides the Perception and Planning which keeps the rover in motion and avoids safety hazards such as obstacles.

   - Knowledge: Perception produces detailed elevation maps via a series of image transformations.

   - Knowledge: Planning evaluates paths by a collection of evaluation functions.

   - Secret: It needs to access the sensor data, goal, maximum allowable velocity, current pose, terrain data maps, etc.

   - Secret: More elaboration can be found in other subsections which particularly address the Navigation system architectures.

3. Map Manager:

   - Map Manager provides a set of terrain maps which may be used in the planning tasks by the Navigation system or to satisfy the requests from users.

   - Knowledge: It knows how to merge multiple maps into a composite one.

   - Secret: It handles both local elevation maps and global DTE maps. Global DTE maps are not changed throughout the rover's mission. We can use "layering" techniques.

   - Secret: Based on some purging criteria, it will purge the map database if memory is full. To apply the purging criteria, it may need to access the current pose, the goal, the "odometer" reading, etc.

4. Reflexive Behavior:

   - Reflexive Behavior continually checks for exceptional situations and responds to these situations.

   - Knowledge: It knows the safety constraints of the rover.

   - Secret: It monitors the power conditions, tilts, temperatures, failures of the other components, etc.

- Secret: It can have different strategies to handle different exceptions. Currently it simply says "stop" to the Real-Time Controller (via Central Repository) and raise warnings to the user (via Central Repository and Mission Manager).

5. Mission Manager:

- Mission Manager is the only component connected to the ground system on Earth, where User Interface is physically allocated.

- Mission Manager receives commands issued from User Interface and responds to requests as needed, such as start/stop experiments, turn on/off color camera, etc.

- Mission Manager may transfer archived data back to the ground station.

- Secret: It controls the modes of movements. One critical command is to change the rover's operational mode (i.e., Autonomous, Teleoperation, Supervised-Teleoperation). Mission Manager responds by modifying the "Mode Control Information" stored in the Central Repository. Other components can consult this information in deciding appropriate behavior while performing their tasks. This information can also be used to coordinate components (please see the description about Coordinator).

6. User Interface:

- User Interface provides access for users to control the rover and obtain status.

- User Interface is the only component which has no access to the Central Repository because it is physically located in the ground station on Earth.

- Knowledge: It owns the drivers of screens, various pointing devices, etc.

- Secret: Hardware related details are hidden.

7. Experiments:

- This is a reserved cell that could run "self-contained" experiments. Presumably an experiment would be commissioned by a scientist.

- Knowledge: It has domain related knowledge as how to conduct an experiment.

- Secret: The experiments have little interaction with the Rover system itself, but we need to control the start/stop of the experiments, control parameters of the experiments, view experiment data, etc.

8. Position Estimation:

- Position Estimation provides the current pose of the rover.

- Knowledge: It owns the knowledge about the rover's kinematic model.

- Secret: We can use the Dead Reckoning algorithm to calculate the current pose of the rover.

9. Real-Time Controller:

- Real-Time Controller provides sensor and motor controls which interface directly with the hardware.

- Knowledge: It owns drivers for each device.
- Secret: It hides all hardware related details.

10. Others:

- Other expert components may be identified as needed.

11. Coordinator:

- Coordinator orchestrates the overall execution of the system. Please see the following subsections for a more complete description.
- It provides access controls among various components.
- Secret: It coordinates components based on the "Mode Control Information" stored in the Central Repository. Since the "Mode Control Information" can be dynamically changed by Mission Manager, the resultant coordination can also be dynamically reconfigured.
- Secret: It serves as a lock manager. If there are multiple parties which contend to read or write data in the Central Repository, it may or may not allow simultaneous data reading, but only allow exclusive data writing.
- Secret: It serves as a prioritizer. When there are multiple pending components waiting for accessing the data, it may decide an access order by prioritizing them according to their relative importance based on the mode. A simple example is given in the Operational Scenario subsection.
- Secret: If one component takes a long time to access data (perhaps because of death or bulk data), or if a very critical component needs to access data (e.g., Reflexive Behavior), Coordinator may need to provide a kind of pre-emption mechanism.

2.1.3 Connectors

In this architecture, there are three kinds of connectors:

1. Data Access:

   The connectors between each expert component and Central Repository denote *data access.* One direction denotes *data retrieval;* the other denotes *data modification.* (If Central Repository is implemented as abstract data types, these connectors could be common procedure/method invocations or remote procedure calls.)

2. Input/Output Communication:

   Several components communicate directly with outside world. Navigation system has stereo data as input from black/white cameras; these stereo data have high bandwidth. Mission Manager receives user commands from and produces responses to User Interface; these data may also have high bandwidth. Real-Time Controller commands actuators and samples sensors; these interactions involve different hardware devices.

3. Access Request/Access Grant:

   These connectors are between each expert component and Coordinator. Each expert component must gain access permission before it can actually access the repository data. It relays its access request to Coordinator. Coordinator will grant the access request if the request is permitted. Once receiving the access grant, the individual component can read and/or write the data in the repository.

2.1.4 Operational Scenario

When in operation, all expert components execute concurrently and actively. For example, Navigation system continuously extracts the goals from the repository, consults elevation maps from Map Database or its own Perception subcomponent, creates waypoints to achieves the goals, and posts motion commands back to the repository. Real-Time Controller continuously extracts motion commands for actuating motors, and posts sensor samples to the repository. Reflexive Behavior continuously examines repository, and posts exception handling command (so far only "stop") to the repository. Mission Manager continuously receives user commands, and changes "Mode Control Information" if necessary. Position Estimation continuously reckons the rover's current pose and posts it back to repository for use by everyone.

In this architectural design, Coordinator plays an important part of the system. It provides access controls among various components; otherwise simultaneous access to Central Repository may result in chaos. It coordinates components based on the "Mode Control Information" stored in the Central Repository. It serves as a lock manager to prevent simultaneous data modifications. It serves as a prioritizer to decide an access order when there are multiple pending requesting agents. It also provides a preemption mechanism.

We can imagine that the design of Coordinator is a big challenge. Some coordination rules can be easily identified:

1. Reflexive Behavior mostly takes the highest priority to access repository data.

2. When in Teleoperation mode, requests from Reflexive Behavior can be delayed (because we better trust the operator than a naive user). In other cases, Reflexive Behavior always preempts others.

3. When in Teleoperation mode, requests for accessing the goal from Navigation can be ignored (because Planning does not need to function in this mode).

However, rules for more sophisticated conditions (such as at least how often the current pose needs to be updated, the relative importance between Real-Time Controller and Map Manager, etc.) need further extensive analysis.

2.2 Rover System 2 - Event-Based Architecture

2.2.1 Introduction

Our second architectural proposal of the rover system is based on an implicit-invocation event system. This view of the system emphasizes encapsulation of domain knowledge into a number of event-driven experts. An event manager determines how events are bound to implicitly-invoked routines provided by each of the experts. The architectural style of the system is depicted in Figure 2.2. Experts exist in the system as components and the primary connection mechanism between experts is event delivery.

An event-based architecture is justified by previous robotics work. Simmons [1] describes event-driven robots that emphasize reaction and do little planning. On the other end of the spectrum are deliberative robots which emphasize planning at the expense of reacting to changes in the surrounding environment. Hybrid systems also described by Simmons have an event-based architectural style where reactive behavior is combined with deliberative planning components.

The rover's physical construction also supports the notion of a distributed event system. Individual hardware components and subsystems abstract away the details of hardware interfaces, electronic signalling, interpretation of video data, and the like. In such a system, it makes sense to have deliberative components in an architecture which, by the style of its design, responds in a preemptive fashion to important asynchronous events like changes in the outside world. The choice of an event-system can be based solely on this criterion. Safety is paramount; it is far more important for the system to react to dangerous situations.

An event-based system is further justified because it offers a high degree of configuration flexibility. Because the system must support many different mission styles (teleoperation, supervised teleoperation, autonomous, patterned), it is desirable to utilize a system architecture which supports reconfiguration without recompilation. The system's mission manager component can configure the event manager's event bindings differently for each mission type.

The mission manager also starts, stops, and configures the system's other components (setting thresholds, parameters, etc.) for each mission type. For example, the navigation component need not operate during teleoperation mode or supervised teleoperation mode, and the teleoperation component need not operate during an autonomous mission. This flexibility may have side effects like increased performance for "lightweight" missions like teleoperation.
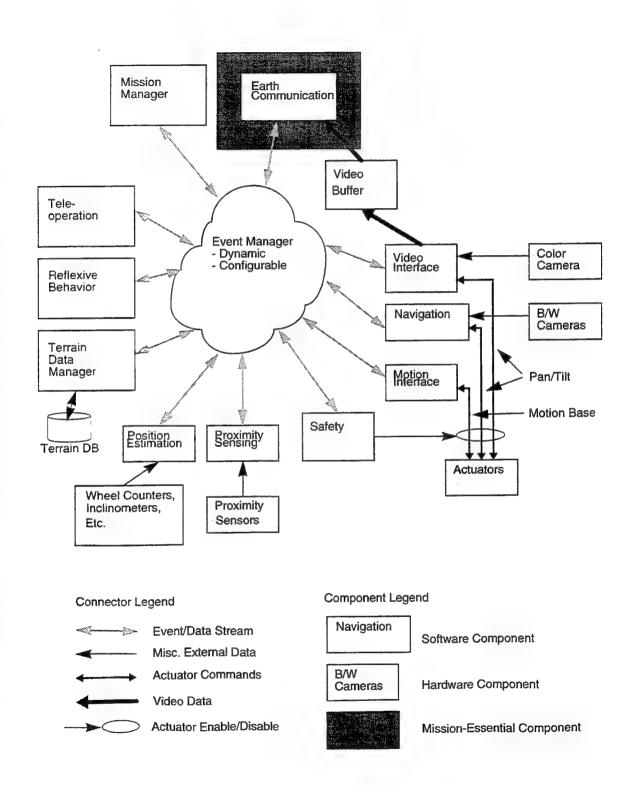
The state of the system is distributed among the domain experts. Each domain expert is specialized at performing a specific task. Most domain experts are responsible for controlling one or many hardware item(s). Each domain expert keeps some information hidden (or secret) and shares other information with other components by generating events.

The system's components communicate with each other through events. Events usually contain attached data, but there may be events which do not communicate data. Likewise, there are different flavors of communication: actual data, references to data in shared memory or video memory, etc.

Because event-based systems are asynchronous by definition, the ordering of event delivery and the subsequent implicit invocation are not guaranteed. There are cases where sequential event sequences are required. To support this, the system must be implemented with transaction mechanisms (transaction identifiers, locking, etc.) that provide transaction-like ACID properties so that sequentiality can be implemented where needed. The mission manager provides these services to the rover's other components.

The architecture described in this section explicitly excludes the user interface. While the user interface is an important component of the system, the overall system's interesting architectural issues regard the design of the system's remote components.

Figure 2-2 Rover System Event-Based Architecture

Figure 2-2 Rover System Event-Based Architecture

Connector Legend

- Event/Data Stream
- Misc. External Data
- Actuator Commands
- Video Data
- Actuator Enable/Disable

Component Legend

- Navigation — Software Component
- B/W Cameras — Hardware Component
- Mission-Essential Component

2.2.2 Components

This section describes each of the rover's components. Each description includes a general overview, identification of information that the component hides from other components, and a summary of the kinds of events each component reacts to and generates.

1. Event Manager (EM):

   - The EM component binds events to component interfaces. It is configured by events sent by the Mission Manager. The actual binding to component interfaces may be procedure call or RPC-based, depending upon the actual implementation.
   - Knowledge: It knows the binding of event types to component interfaces.
   - Secret: The current bindings of events to component/routine name.
   - EM receives events which configure event/response bindings.
   - EM sends periodic events to report its status/health.

2. Mission Manager (MM):

   - The MM component configures the system for operation, monitors other component's progress or health, reacts to their failures, and shuts down other components at the end of a mission. Key to the MM's configuration process is establishing the values of other component's control parameters. The MM dynamically configures the EM's event bindings according to the kind of mission being performed.
   - Knowledge: MM knows the configuration of each mission type and the corresponding configuration parameters for all other components. MM provides transaction-like event-identification and locking mechanisms for components which require sequential ordering of event sequences. As such, it keeps a log of active and recently-completed event sequences for fault recovery.
   - MM receives events regarding selected mission changes, and other events regarding the status or health of other components. It handles requests for transaction IDs, results of transaction commitment, and requests for transaction status.
   - MM sends events which control and configure other components, and events regarding transaction mechanisms.

3. Earth Communications (EC):

   - The EC component provides the rover's link to Earth. As such, and because it starts the Event Manager and Mission Manager upon initial boot, it is the component which has the highest reliability requirements. It must be very fault-tolerant and be able to restart itself in error situations, have contingency strategies for dealing with loss of Earth communications, and ensure the efficient operation of the available communications bandwidth.
   - Knowledge: EC knows the current state of communications with Earth.

- Secret: the hardware and algorithms required to maintain communications with Earth.

- EC receives events regarding mission status, acks and nacks of commands sent from Earth, telemetry events, and data events ("data is available at shared location X" or "here is data Y"). It also receives configuration events.

- EC sends events which correspond to commands from Earth. It generates periodic status/health events.

4. Teleoperation:

- Teleoperation translates teleoperation directives from Earth into motion commands which turn the rover's wheels, pan and tilt the rover's color camera, etc.

- Knowledge: It owns the algorithm which translates teleoperation directives into motion directives. It is the expert that addresses issues that are created by communication delays to and from Earth.

- Teleoperation receives teleoperation directive events and configuration events.

- Teleoperation sends motion request events (turn-wheels) and pan/tilt requests. It generates periodic status/health events.

5. Reflexive Behavior (RB):

- RB provides motion commands, mission change commands, and safety commands in response to events generated by other components. It is active in determining if safety-critical thresholds have been exceeded. If so, it generates events and event sequences which keep the rover from getting into a worse situation, or take predetermined action(s) to decrease the immediate safety risk. It uses the current motion directive to determine if the rover is going to wedge itself into an undesirable position.

- Knowledge: RB owns a set of reflexive behavior sequences and the rules for determining when to apply them.

- RB receives events which report dangerous situations and configuration events. RB also receives motion directives.

- RB generates motion directives, events which request the deactivation and reactivation of actuator command paths, and periodic status/health events.

6. Terrain Data Manager (TDM):

- The TDM is a safe store for maps used by the navigation component. It is not a subcomponent of the navigation system because maps need to be transmitted to and from Earth. This component may use a specialized connector because of the large amount of data being sent to and from it.

- While the actual database is depicted in the architectural diagram as a disk drive, logistics considerations will rule out the use of rotating media for non-Earth mission deployments. In all likelihood, this will be some memory array which will be inherently size-limited. TDM, therefore, will be an active participant in choosing exactly which map fragments are to be on-board the rover at any given time.

- Knowledge: It owns the rover's maps and optimized methods for accessing them.

- TDM receives update_map, get_map events, and configuration events.

- TDM sends map_updated (ack of update_map) and map_contents events. It generates periodic status/health events.

7. Position Estimation:

- Position Estimation uses external signals from wheel counters, inclinometers, and other sensors to determine the rover's exact position and pose. It periodically generates an event which notifies other components of the rover's position and pose.

- Knowledge: It owns the hardware interface to the rover's inclinometers, wheel counters, etc. It encapsulates algorithms used for dead reckoning the rover's current location.

- Receives set_rover_pose events (to baseline the current position) and configuration events.

- Periodically sends rover_pose events. It generates periodic status/health events.

8. Proximity Sensing (PS):

- The proximity sensing component provides a hardware interface to the rover's proximity sensors. It notifies other components when an object is within range of any of the sensors.

- Knowledge: It owns the hardware interface that communicates with proximity sensors.

- PS receives configuration events.

- PS sends events that signal when an obstacle is within sensor range. It generates periodic status/health events.

9. Safety:

- The Safety component provides part of the functionality identified by the customer as the "real-time controller". If a dangerous situation is reported, and part of the response is to disable motion actuator controls, this component actively disables the transmission of actuator commands. It must likewise be able to reactivate such transmissions.

- Knowledge: It owns the hardware which enables and disables actuator commands. It does not disable/enable data transmission from external devices which are incapable of affecting the safety characteristics of the rover's pose, e.g. position estimating sensors, camera data (video), and the like.

- Safety receives events which request the deactivation and activation of actuator command paths and events which request the status of actuator command paths. It also receives configuration events.

- Safety sends acknowledgment (ack/nack) events and responses to status requests. It generates periodic status/health events.

10. Motion Interface (MI):

- Motion Interface provides the interface to the rover's motion base. The component ensures that motion directives are translated into hardware-specific motion commands which are executed by the motion base hardware. Some directives may require an acknowledgment (e.g. "stop").

- Knowledge: It owns the hardware interface to the rover's motion base.

- MI receives motion directive and configuration events.

- MI sends ack/nack events for those requests which require a response. It generates periodic status/health events.

11. Navigation:

- Navigation transforms goals into motion directives. A goal can describe a desired location, a pattern of motion to accomplish, or an area of locations to visit. Once a plan has been established and is being enacted, Navigation ensures that the plan is being enacted within tolerances described by some threshold. Navigation also responds to requests for maps of a specific areas. The navigation component is provided with the rover's current pose as generated by Position Estimation.

- Knowledge: It owns the algorithms used to perceive the outside world (perception), and algorithms for determining route and path selection. Navigation controls the rover's hardware that is used to perceive the outside world, i.e. the pan/tilt mechanism for the black & white stereo cameras. This set of hardware does not include the rover's color camera.

- Secret: The current plan(s) and thresholds associated with the current mission.

- Navigation receives goal events, threshold selection events, and configuration events.

- Navigation sends motion directive events, map request events (satisfied by TDM), and mission status events (e.g. got_there, can't_get_there, etc.). It generates periodic status/health events.

12. Video Interface (VI):

- Video Interface provides the capability to grab images from the color camera for transmission back to Earth. Its primary use is for teleoperation missions, but VI can be configured into any mission. VI places grabbed images into the Video Buffer. VI then generates an event regarding the location of the grabbed image.

- Knowledge: It owns the hardware interface that controls the color camera's pan/tilt mechanism.

- VI receives configuration events (e.g. frame grab rate) and camera pan/tilt request events.

- VI sends events announcing the availability of images in the Video Buffer. It generates periodic status/health events.

2.2.3 Connectors

Each of the next paragraphs describe a unique connector type.

1. Event/Data Stream Connectors - These connectors move events to and from the system's components. There are a number of "colors" of event connectors: The first kind of connector communicates only event notifications. The second kind of event adds actual data to the notification. The last kind of event connector contains the event notification and a reference to shared data. An example of the latter connector is the event stream that VI sends to EC regarding images which have been placed into Video Buffer.

2. Video Data Connectors - This kind of connector is specialized for transmitting video frames from the rover to Earth. The Video Interface component grabs image, places them into video memory, and notifies Earth Communications (EC) of the location of the image. EC then uses the event information to transmit the image (under constraints that EC hold as secret) back to Earth. VI and EC must coordinate access to the video memory to avoid consistency problems. This can be done by constructing a coordinated event sequence using transaction mechanisms provided by the Mission Manager.

3. Actuator Commands Connectors - These commands and protocols are device-specific. The architectural design of the system encapsulates each of the device-specific details in a different module. This will not have been a good decision if the devices use very common, homogeneous actuator interface. Such a homogeneous interface would suggest a common actuator interface.

4. Miscellaneous External Data Connectors - Likewise, these data streams and protocols are device-specific. Some protocols will have a poll-response flavor while others (e.g. proximity sensors) will be driven by discrete events.

2.2.4 Operational Scenario

Once the rover has been deposited at some location on some terrestrial orb (the Earth or Moon), the machine is powered on by some external action. The communications interface component begins its boot process. Once complete, the component is as a partner in establishing communications with Earth. In the meantime, the Earth Communications cold-bootstraps the Event Manager and the Mission Manager. It then waits for its next command from Earth.

The Mission Manger, once started, establishes its link with the Event Manager and configures the Event Manager to some initial state. In this state, mission change events invoke methods in the Mission Manager and transmission requests will invoke methods in Earth Communications.

At this point, the rover system is in the "sleep" state as defined in the system specification and is ready to begin normal mission operations.

From this point on, the Mission Manager can start, operate, pause, and tear down mission configurations. The Mission Manager, Event Manager, and Earth Communications components are the only long-lived components that are operating throughout this process. All other components are in a sense transient, and are only in existence to support Earth-directed mission selections.

A mission selection, therefore, has the following life:

- A mission is selected. The Mission Manager starts components, configures event bindings, and sends configuration events to components.
- The components operate according to the configuration in order to accomplish the mission. The mission may be paused (sleep mode) and optionally restarted.
- The mission is stopped. The Mission Manager stops the mission's components, reconfigures event bindings to accept a new mission selection, and waits for the next selection from Earth.

The following mission selection descriptions provide an architectural-level identification of components that comprise a mission and an incomplete list of component configuration suggestions. These few examples strive to show why the concept of dynamic event bindings is relevant to the rover's architecture.

Teleoperation:

- Components: All except Navigation, Position Estimation, and Terrain Data Manager.
- Configuration Examples: Reflexive behavior parameters reduced to minimums as a well-trained expert is driving the rover.

Supervised Teleoperation:

- Components: All except Navigation and Terrain Data Manager.
- Configuration Examples: Reflexive Behavior parameters adjusted for maximum rover safety as anyone could be driving the rover.

Autonomous/Patterned Motion (Normal Operation):

- Components: All except Teleoperation and Video Interface.
- Configuration Examples: All status/health reporting at maximums.

Autonomous/Patterned Motion (testing new Navigation Algorithm):

- Components: All except Teleoperation.
- Configuration Examples: All status/health reporting at maximums. Reflexive Behavior thresholds set for maximum safety.

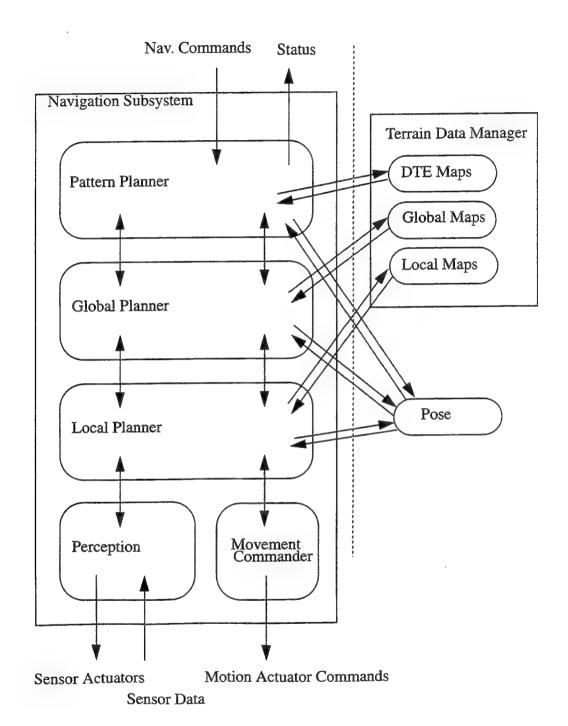## 2.3 Navigation Subsystem 1 - Layered Architecture

2.3.1 Introduction

The system depicted below is an architectural proposal of the Navigation subsystem as a layered system. The planning tasks performed by Navigation system are broken down into different stages that process the information until specific commands for the actuators can be executed.

Tasks are defined as different stages in which a plan can be broken into. This leaves us with four groups of tasks each of which performs different computations. The three planning components consult the map structures in different levels of detail. This is achieved by enquiring data from Terrain Data Manager.

Data flows horizontally from the tasks to the appropriate map structures and the pose structure. Control flows vertically form one layer to any of its consecutive counterparts. This essentially defines the layered property of the system. There is no direct control flow between any two non-adjacent layers.

This organizational approach is motivated by the simplicity of the encapsulation that the system lends itself to. The tasks of Navigation system can be naturally divided into a series of levels of refinement, and each of these levels deals with the processing in different levels of resolution. The detailed maps that the Navigation subsystem creates and utilizes may also be accessible to other components outside the Navigation subsystem. This is achieved by transmitting maps to the Terrain Data Manager. Outside components can therefore request maps from the Navigation subsystem where Perception will transform sensor data and produce detailed maps.

Figure 2-3 Navigation Subsystem Layered Architecture

2.3.2 Components

1. Pattern Planner:

   - Pattern Planner receives the external navigation commands and processes them into intermediate goals that describe the pattern specified by the commands. It will then relay these intermediate goals to the Global Planner. Global Planner returns the current status of the operation. Pattern Planner will send the status information back as an Acknowledgment, or in the case that the operation cannot be accomplished, as a Failure.

   - Knowledge: It knows how to generate a sequence of waypoints which satisfies the designated pattern.

   - Secret: It must use the current pose and the DTE maps.

2. Global Planner

   - Global Planner creates directives to take the rover from one intermediate goal to the next.

   - Knowledge: It can create a sequence of immediate goals to achieve an intermediate goal, It can also handle map request.

   - Secret: It must use the current pose and the global maps.

3. Local Planner

   - Local Planner handles an immediate goal based on the elevation maps computed by Perception. It will send a selected trajectory to the Movement Commander.

   - Knowledge: Given an immediate goal, it can evaluate a set of potential trajectories and select the most promising one, to try to attain the immediate goal.It is also able to handle map requests.

   - Knowledge: It knows the maximum allowable velocity and other safety violation conditions.

   - Secret: It must use the current pose. It must use the detailed elevation maps. It will evaluate trajectories using a collection of evaluation functions.

4. Perception

   - Perception generates commands to the Sensor Actuators, computes the detailed elevation maps, and relays them to the Local Planner.

   - Knowledge: It owns algorithms to generate Sensor Actuator directives and it knows how to interpret the sensor data.

   - Secret: It calculates a detailed elevation map through a series of image processing transformations.

5. Movement Commander

   - It receives directives from the Local Planner. It translates them into commands that are send to the Motion Actuators (outside Navigation subsystem).

2.3.3 Connectors

1. The system is connected to the outside world at all layers. The interaction might be done in the form of messages. Pattern Planner receives navigation commands from the rover system and returns acknowledgments. Global Planner interacts with the global maps of the Terrain Data Manager. Local Planner interacts with the local maps of the Terrain Data Manager. All the planning layers need to obtain information about the current pose. Movement Commander sends motion commands back to the rover system for execution.

2. Perception will send commands to the camera actuators. Perception will read sensor data for processing. These kinds of interaction involve access to the hardware devices.

3. Communication between the layers is bi-directional and might be done through procedure calls.

2.3.4 Operational Scenario

On a typical scenario, the Pattern Planner receives a navigation command from the rover system, and generates a set of intermediate goals that correspond to the pattern specified in the command if any. For each of these intermediate goals, it will call the Global Planner for each intermediate goal. The Global Planner will generate immediate goals and pass the immediate goal to the Local Planner. The Local Planner will evaluate potential trajectories and select one passable trajectory.

The selected trajectory is passed to the Movement Commander. The Movement Commander generates a motion command based on the selected trajectory, and sends the motion command back to the rover system for execution.

The Local Planner will need the elevation maps computed by Perception. Perception calculates maps from stereo data. Once the Local Planner achieves the immediate goal, it will return to the Global Planner and wait for the next immediate goal. The Global Planner will request and maintain its own maps, and return to the Pattern Planner once it has reached its intermediate goal. The Pattern Planner will keep passing intermediate goals until the pattern is completed and it can send an Acknowledgment back to the rover system.

A number of conditions will be checked at each of the layers to know if the navigation command can be realized. The Navigation system described will not however be able to handle any emergency situation. The Navigation system therefore should be shut down and its actions be taken over by a reflexive module when in an emergency.

2.4 Navigation Subsystem 2 - Blackboard Architecture

2.4.1 Introduction

The Navigation subsystem provides perception and planning support to the main rover system. In general, planning and perception activities require the following inputs from the main rover system:

- goals
- patterns
- regions
- current pose
- maximum velocity
- map requests
- terrain maps

Upon evaluation of the above inputs, the Navigation subsystem performs the appropriate planning and perception activities which result in the following outputs:

- motion commands
- map data
- planning status

The Navigation subsystem consists of a blackboard and several knowledge sources.

The blackboard contains the current state of planning, perception, and the pose. This data is used by knowledge sources to reason about and perform perception and planning tasks. The blackboard also provides a workspace which knowledge sources can use to post intermediate and final results. Furthermore, the blackboard provides the medium through which components communicate with each other and the main rover system.

The blackboard is partitioned into levels which are distinguished from one another by the specific types of data they can store. Knowledge sources have read and write access to various levels depending on the tasks they perform, the data they require, and the data they output. When a level receives new data or has existing data updated, the knowledge sources that rely on data from that level are notified. When data on a level has been added or changed, each knowledge source evaluates the data and decides whether or not to respond.
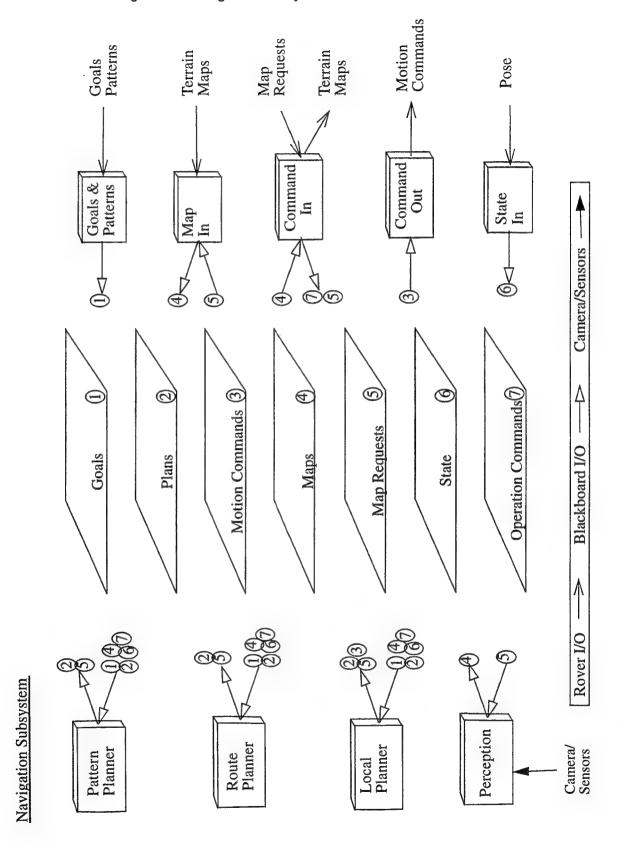
Knowledge sources have specialized knowledge which depends on the tasks they perform. In particular, knowledge sources provide the ability of the navigation subsystem to communication with the main rover system, to perform local, route, and pattern planning, and to perform perception tasks. There are five knowledge sources which provide Navigation's ability to communicate with the main rover system, namely: the goals-and-patterns-ks, map-in-ks, command-in-ks, command-out-ks, and state-in-ks. In addition, there are three knowledge sources

which provide Navigation's ability to reason about and perform planning activities, namely: the pattern-planner-ks, route-planner-ks, and local-planner-ks. Finally, there is one knowledge source which provides Navigation's perception capabilities.

Knowledge sources were selected based on the need to encapsulate the details of planning, communication, and perception and separate concerns to best support system evolution.

The remaining subsections provide more details about the blackboard and knowledge sources. Details are also provided regarding the internal connections required for knowledge sources to communicate with each other and the external connections required for knowledge sources to communicate with the main rover system and support hardware. The final subsection provides operational scenarios which describe how knowledge sources work together to perform planning and perception tasks.

Figure 2-4 Navigation Subsystem Blackboard Architecture

2.4.2 Components

1. Blackboard

   - Level 1:

     This level is used to post goals and patterns provided by the main rover system.

   - Level 2:

     This level is used to post route, pattern, and local plans. Partial plans and potential routes are also posted here for evaluation and refinement by various knowledge sources.

   - Level 3:

     This level is used to post motion commands and status messages such as 'stuck' for transmission to the main rover system.

   - Level 4:

     This level is used to post detailed elevation maps and DTE maps.

   - Level 5:

     This level is used to post requests for maps and terrain data.

   - Level 6:

     This level is used to post pose and current speed data provided by the main rover system.

   - Level 7:

     This level is used to post commands such as 'stop planning' from the main rover system.

2. Pattern Planner

   - The main task of the pattern-planner-ks is to provide coarse-grained plans which can be used to direct the rover through a region according to a given pattern.

     Pattern planning activities are initiated in response to set-pattern messages posted to level 1 and plans posted to level 2.

     The pattern-planner-ks uses information on levels 6 and 4 while generating plans. Level 6 provides information about the rover's current state and pose. Level 4 provides current map and terrain data. If the pattern-planner-ks requires map data that is not available on level 4 then it will post one or more map requests on level 5. The pattern-planner-ks will then wait for corresponding map data to be posted on level 4.

     Plans created by the pattern-planner-ks are posted on level 2. The pattern-planner-ks may also modify plans on level 2 according to newly established pattern goals.

     The pattern-planner-ks also halts planning activities in response to stop messages posted on level 7.

3. Route Planner

- The main task of the route-planner-ks is to provide coarse-grained plans which can be used to direct the rover from its current position to a given goal position.

  Route planning activities are initiated in response to set-distant-goal messages posted to level 1 and plans posted to level 2.

  The route-planner-ks uses information on levels 6 and 4 while generating plans. Level 6 provides information about the rover's current state and pose. Level 4 provides current map and terrain data. If the route-planner-ks requires map data that is not available on level 4 then it will post one or more map requests on level 5. The route-planner-ks will then wait for corresponding map data to be posted on level 4.

  Plans created by the route-planner-ks are posted on level 2. The route-planner-ks may also modify plans on level 2 according to newly established distant goal messages.

  The route-planner-ks also halts planning activities in response to stop messages posted on level 7.

4. Local Planner

- The main task of the local-planner-ks is to refine coarse-grained plans posted on level 2 and to carry out those plans by issuing commands to the main rover system.

  Local planning activities are initiated in response to set-goal messages posted on level 1 or in response to coarse-grained pattern or route plans posted on level 2.

  The local-planner-ks uses information on levels 6 and 4 while generating plans. Level 6 provides information about the rover's current state and pose. Level 4 provides current map and terrain data. If the local-planner-ks requires map data that is not available on level 4 then it will post one or more map requests on level 5. The local-planner-ks will then wait for corresponding map data to be posted on level 4.

  The local-planner-ks refines a coarse-grained plans by considering its major waypoints and then generating potential paths between those waypoints. The local-planner-ks then determines the best potential path and accordingly refines the coarse-grained plan.

  Plans refinements created by the local-planner-ks are posted on level 2. The local-planner-ks may also modify plans on level 2 according to newly established goals or current circumstances.

  Upon completing the evaluation of a plan, the local-planner-ks computes the motion commands required to carry out the most promising path and post those commands to level 3.

  The local-planner-ks also halts planning activities in response to stop messages posted on level 7.

5. Perception

- The main task of the perception-ks is to provide map and terrain data.

  Perception activities are initiated in response to map requests posted to level 5.

  The perception-ks will attempt to post data which satisfies a map request on level 4. The perception-ks will update requests on level 5 to indicate when they cannot be satisfied.

- Secret: Encapsulates how sensor data is input and processed. Also, hides the details of camera manipulation, camera input, and camera data translation.

6. Goal and Pattern

- The main task of the goal-and-pattern ks is to input goal and pattern requests from the main rover system.

  The goal-and-pattern-ks responds to input goals or patterns by parsing them and then posting them to level 1.

- Secret: Encapsulates how goals and patterns are input from the main rover system. Also translates input goals and patterns into a form which is recognized by peer components.

7. Map In

- The main task of the map-in-ks is to request map data from the main rover system.

  Requests for map data are initiated in response to unsatisfied map requests posted to level 5.

  The map-in-ks requests map data from the main rover system and waits for a response. Any data received is posted on level 4. The map-in-ks will update requests on level 5 to indicate when they cannot be satisfied.

- Secret: Encapsulates how map data is acquired from the main rover system. Also hides any differences between the map representation used by the main rover system and the navigation subsystem.

8. Command In

- The main task of the command-in-ks is to input and perform commands from the main rover system.

  The command-in-ks responds to stop commands by posting stop requests to level 7.

  The command-in-ks responds to map requests by transmitting the requested map data to the main rover system. The data is acquired from level 4. If the data is not available on level 4 then the command-in-ks posts a map request to level 5 and waits for the corresponding data to be posted to level 4. If the request cannot be satisfied then the command-in-ks informs the main rover system.

- Secret: Encapsulates how commands are input from the main rover system. Also translates input commands into a form which is recognized by peer components.

9. Command Out

- The main task of the command-out-ks is to output motion commands and status messages to the main rover system.

  Output to the main rover system is initiated in response to motion commands or statuses posted to level 3. Motion commands will include 'Stop Rover'. Status commands will include 'Stuck'.

- Secret: Encapsulates how motion commands are output to the main rover system. Also handles translation of logical motion commands into commands required to manipulate the rover.

10. State In

- The main task of the state-in-ks is to parse state and pose data provided by the main rover system and to post that data on level 6.

- Secret: Encapsulates how state data is acquired from the main rover system. Also handles translation of state information into a form which is recognized by peer components.

2.4.3 Connectors

The Navigation subsystem requires the following three types of connectors:

- Input/Output to the main rover system

  Provides the means for communication between Navigation and the main rover system. Also establishes communication protocols and data translations.

  This could be implemented via implicit or explicit procedure call.

- Knowledge source access to the blackboard

  Provides read/write access to individual blackboard levels. Also establishes the protocols and data translations for the types of data that can be posted on each level.

  This could be implemented via shared memory access, and implicit or explicit procedure call.

- Perception link to camera and sensors

  Provides the means for accessing sensor data, manipulating the camera, and acquiring camera data. Also establishes the protocols and data translations which support these tasks.

  This could be implemented via shared memory access, interrupts, or explicit procedure call.

2.4.4 Operational Scenario

1. Stop Request

   The command-in-ks receives a stop command from the main rover system. The message is evaluated and posted on level 7.

   The local planner, the route planner, and the pattern planner all respond to the stop command by halting planning activities.

2. Set Goal Message

   The goal-and-pattern-ks receives a set-goal message from the main rover system. The message is evaluated and posted on level 1.

   The local-planner-ks evaluates the goal message posted on level 1 along with state and position information posted on level 6. If the goal has already been met then the local-planner-ks posts a stop command on level 3. The command-out-ks responds to the stop command on level 3 by outputting a stop command to the main rover system.

   If the goal has not been met then the local-planner-ks will attempt to develop potential paths for reaching the goal. Potential paths are posted to level 2 where they are refined or eliminated by the local-planner-ks or other knowledge sources.

   The local-planner-ks continues to refine and eliminate potential paths to the goal until one path can be selected. Afterwards, the local-planner-ks computes the required motion commands and posts them to level 5. The command-out-ks responds to motion commands posted on level 5 by outputting them to the main rover system.

   If the local-planner-ks cannot find an acceptable path to the goal then it posts a stuck message on level 5. The command-out-ks responds to the stuck message by outputting the message to the main rover system.

   While evaluating a goal and generating potential paths for reaching the goal, the local-planner-ks may use map data posted on level 4. If local-planner-ks requires map data but cannot find it on level 4 then it will post one or more map requests to level 5. Any map requests posted to level 5 are processed by the perception-ks or the map-in-ks. If possible, the perception-ks will post the requested map data to level 4. Otherwise, it will update any map requests to show that they could not be satisfied. The map-in-ks responds to unsatisfied map requests by requesting map data from the main rover system. Once it receives the map data, the map-in-ks posts the data on level 4.

3. Set Distant Goal Message

   The goal-and-pattern-ks receives a set-distant-goal message from the main rover system. The message is evaluated and posted on level 1.

   The route-planner-ks evaluates the goal message posted on level 1 along with state and position information on level 6. If the goal has already been met then the route-planner-ks does not take any action.

The local-planner-ks also evaluates the goal message posted on level 1. If the goal has already been met then the local-planner-ks posts a stop command on level 3. The command-out-ks responds to the stop command on level 3 by outputting a stop command to the main rover system.

If the goal has not been met then the route-planner-ks will develop a coarse-grained plan to reach the goal. The plan will be posted on level 2. Afterwards, the plan will be evaluated and refined by the local-planner.

Once the plan has been adequately refined, the local-planner-ks computes the required motion commands and posts them to level 5. The command-out-ks responds to the motion commands posted on level 5 by outputting them to the main rover system.

If the local-planner-ks cannot adequately refine the plan then it posts a stuck message on level 5. The command-out-ks responds to the stuck message by outputting the message to the main rover system.

While developing or refining plans, the local-planner-ks and the route-planner-ks may use map data posted on level 4. If a knowledge source requires map data but cannot find it on level 4 then it may post one or more map requests to level 5. Any map requests posted to level 5 are processed by the perception-ks or the map-in-ks. If possible, the perception-ks will post the requested map data to level 4. Otherwise, it will update any map requests to show that they could not be satisfied. The map-in-ks responds to unsatisfied map requests by requesting map data from the main rover system. Once it receives the map data, the map-in-ks posts the data on level 4.

4. Set Pattern Message

The goal-and-pattern-ks receives a set-pattern message from the main rover system. The message is evaluated and posted on level 1.

The pattern-planner-ks responds to the set-pattern message by developing a coarse-grained plan to reach the goal via the target pattern. The plan will be posted on level 2. Afterwards, the plan will be evaluated and refined by the local-planner.

Once the plan has been adequately refined, the local-planner-ks computes the required motion commands and posts them to level 5. The command-out-ks responds to the motion commands posted on level 5 by outputting them to the main rover system.

If the local-planner-ks cannot adequately refine the plan then it posts a stuck message on level 5. The command-out-ks responds to the stuck message by outputting the message to the main rover system.

While developing or refining plans, the local-planner-ks and the pattern-planner-ks may use map data posted on level 4. If a knowledge source requires map data but cannot find it on level 4 then it may post one or more map requests to level 5. Any map requests posted to level 5 are processed by the perception-ks or the map-in-ks. If possible, the perception-ks will post the requested map data to level 4. Otherwise, it will update any map requests

to show that they could not be satisfied. The map-in-ks responds to unsatisfied map requests by requesting map data from the main rover system. Once it receives the map data, the map-in-ks posts the data on level 4.

5. Map Request

The command-in-ks receives a map data request from the main rover system.

If the map data is available on level 4 then the command-in-ks outputs the data to the main rover system.

If the map data is not available on level 4 then the command-in-ks will post one or more map requests on level 5. Map requests posted on level 5 are processed by the perception-ks. If possible, the perception-ks will post the requested map data to level 4. Otherwise, it will update the requests to show that they could not be satisfied. The command-in-ks will notify the main rover system if the map request cannot be satisfied.

3   Design Rules, Choices, and Justifications

Section 3.1 discusses a collection of design rules applied to a general robotic system, and also a collection of design rules applied to the planning tasks underneath a robotic system. Based on these rules and the characteristics of system's functional requirements, section 3.2 discusses and prioritizes our design choices. Section 3.3 discusses how to integrate our designs.

3.1   Design Rules

The use of design space and design rules for software system design was proposed by Thomas G. Lane. The design space identifies the key functional and structural dimensions used to create a system design. We treat our two architectural proposals as one dimension, identify several functional dimensions, and derive some rules relating them.

We discuss two sets of design rules for the general robotic system and the navigation subsystem, respectively. The lists of rules are not intended to be complete, but they provide a sound basis upon which we can make sensible design choices.

3.1.1   Design Rules for Rover System

The following rules are associated with a general robotic system. Note that the last column denotes issues that are relatively more important for the Rover system, but a blank without an asterisk symbol does not imply the issue is not applicable to the Rover system.

Table 1: Design Rules for Rover System

Rule ID	Name	Suitable Architecture	Important for Rover
ROV-1	Distributed processing environment	Event	*
ROV-2	Centralized processing environment	Repository	
ROV-3	Support for adding new components	Event	*
ROV-4	Support for a reconfigurable system	Event	*
ROV-5	Need to ensure a consistent state	Repository	
ROV-6	Ease of performance analysis	Repository	*
ROV-7	Ease of state monitoring	Repository	*
ROV-8	Synchronous processing required	Repository	
ROV-9	Asynchronous processing required	Event	*
ROV-10	Strong need of sequentiality	Repository	
ROV-11	System overhead minimized	Repository	

- ROV-1: In a physically distributed system organization with multiple CPUs and non-negligible communication costs such as in a computer network, it is better to adopt an event-based architecture. A repository requires a shared memory and a state polling mechanism, which are usually expensive to implement in a distributed environment.

- ROV-2: A centralized computing environment favors the repository design. A shared memory and an access mechanism are easier to implement and can achieve relatively high efficiency.

- ROV-3: In an event-based system, new components can register interests by associating their handling routines with events. A component raising an event does not have to know the identity of the other components. However, in our repository design, adding a new component requires a complete redesign of the Coordinator.

- ROV-4: An event system is favored than a repository if the system requires reconfigurable features, especially the dynamic reconfigurations (changeable at run-time). The event manager can change its event bindings at run-time without recompilation. In our repository design, we can only implement static reconfigurations by hard-coating certain configuration types in Coordinator.

- ROV-5: If the consistency of the whole system state is critical for the application, a repository is preferable since all state information is centrally kept in the shared memory. We may simply embed a small routine to perform checking. The system state, however, is essentially distributed among all components in an event-based system. Thus collecting system state information and checking consistency with each other is expensive in an event-based system.

- ROV-6: We contend that system performance is easier to be analyzed in our repository design than in the event-based design. Since the Coordinator is naturally the core agent to implement the preemptive mechanism, performance analysis techniques such as Rate Monotonic Analysis in real-time field can readily be applied in the repository design.

- ROV-7: If the state information is centrally maintained, the monitoring tasks like Reflexive Behavior can be simpler and more efficient. A repository is preferred for the same reason as indicated in ROV-5.

- ROV-8: For an application with intensive synchronization features, a repository is favored. A transaction processing system is a typical example. Those ACID properties are assured by the Coordinator. The Central Repository and the Coordinator together essentially provide the TP system core services such as locking, logging, etc. An event-based system is relatively difficult to implement these properties due to its non-deterministic nature.

- ROV-9: An event-based design is a concurrent system in nature. All components are weakly bound through the event bindings, and therefore individual components are free to asynchronously execute their own tasks at other times. In a repository design, all components tend to contend for accessing the shared data, and blocking is an unavoidable feature. Thus a repository is not appropriate for an asynchronous application.

- ROV-10: In an event-based system, announcers don't know which components will be affected by the events. Thus components cannot make assumptions about the order of processing. In our repository design, the Coordinator explicitly imposes a control discipline and therefore can ensure the property of sequentiality.

- ROV-11: An event-based system usually introduces inevitable system overhead such as the run-time checking and implicit invocation mechanisms. It is even worse if the system cannot ensure circularity free, which might cause deadlocks. Arguably we claim the repository only implements the necessary access control mechanisms, which cost less overhead.

3.1.2 Design Rules for Navigation System

The following rules can be applied to the navigation subsystem within a robotic system, which usually involves both reactive and deliberative planning activities. Note that the last column denotes issues that are relatively more important for the Navigation component, but a blank without an asterisk symbol does not imply the issue is not applicable to the Navigation system.

Table 2: Design Rules for Navigation Subsystem

Rule ID	Name	Suitable Architecture	Important for NAV
NAV-1	Solution to problem is algorithmic	Layered	
NAV-2	Solution to problem is heuristic	Blackboard	*
NAV-3	Opportunistic reasoning required	Blackboard	*
NAV-4	Deterministic task scheduling required	Layered	
NAV-5	Likely to change processing approaches	Layered	*
NAV-6	Likely to add processing approaches	Blackboard	*
NAV-7	Incremental development required	Blackboard	*
NAV-8	High need for predictable performance	Layered	
NAV-9	High need for component independence	Blackboard	
NAV-10	High need for economic solution	Layered	

- NAV-1: If the problem has a precise algorithmic solution, usually a layered system is a good choice. The processing steps can usually be broken down into a hierarchical top-down fashion, with each layer solving subproblems in different details. Processing using the divide-and-conquer strategy is a typical example. It is difficult to naturally implement such a structure in a blackboard system.

- NAV-2: If the problem has no absolutely correct solution and no direct algorithmic solution, usually a blackboard is a good choice. "Best effort" or approximate solution is often good enough. Multiple distinct kinds of heuristic rules can be embedded in different knowledge sources. Knowledge sources can achieve cooperative problem-solving through the blackboard.

- NAV-3: This essentially relates to NAV-2. An application involving much uncertainty favors a blackboard structure. Both input and knowledge have many errors and variabilities, and therefore what kinds of reasoning are applicable is really opportunistic. The blackboard framework does not presuppose nor does it prescribe the knowledge usage or reasoning methods. It merely provides constructs within which any reasoning methods can be well applied.

- NAV-4: If the application requires a deterministic task scheduling discipline, then the layered system is probably a better choice. The thread of execution in a layered system is more likely to be predecided and conceived.

- NAV-5: Comparing a layered system with a blackboard, the hierarchy of layers eases the changes of processing approaches. You are more confident to replace an internal algorithm of a layer without changing the overall behavior of the system. In a blackboard, however, if you change the rules of some knowledge source, the effects will propagate in the solution, and probably the overall behavior of the system is affected.

- NAV-6: Comparing a blackboard with a layered system, the distribution of knowledge sources eases the addition of new processing approaches. An additional collection of heuristic rules or algorithmic procedures can be easily added as a new knowledge source to the blackboard.

- NAV-7: If the nature of the problem prevents from having a full solution all at once, a blackboard is favored. Since each execution can only result in a partial solution, the entire solution needs to evolve over time. The blackboard data structure provides the base for incremental development.

- NAV-8: The time to come up with a solution is usually unpredictable in a blackboard system. Although the hierarchical structure of a layered system imposes certain interaction overhead, the processing time is more likely to be empirically predetermined.

- NAV-9: To achieve high component independence, a blackboard is a better choice. A layered system is vertically related to each other because each layer assumes the services and interfaces of adjacent layers. In a blackboard, however, each knowledge source can operate independently.

- NAV-10: To make a blackboard really work, sophisticated elaboration is needed. For example, an interpreter implementation strategy is needed to impose the control mechanism. If economics are a significant concern, a blackboard architecture is not a good choice.

3.2 Design Choices

3.2.1 Design Choice for Rover System

Considering those important concerns of the Rover system, we prioritize them based on the functional requirements as follows. Their relative importance is shown in order.

Table 3: Prioritizing Design Considerations for Rover System

Name	Suitable Architecture
Asynchronous processing required	Event
Distributed processing environment	Event
Support for a reconfigurable system	Event
Ease of state monitoring	Repository
Support for adding new components	Event
Ease of performance analysis	Repository

The Rover system mostly requires concurrent and asynchronous processing. It is implemented in a distributed environment (probably Ethernet). The rover has different missions and each mission needs a different system configuration. The reflexive behavior requires monitoring the system state to ensure safety, but this kind of monitoring tasks is not highly complicated. The system has medium extensibility requirements (adding more experiments components). The performance is not a very significant issue as is the safety.

Judging from the table (either in terms of numbers or importance), we recommend that the event-based architecture is a better design choice for the Rover system.

3.2.2 Design Choice for Navigation System

Considering those important concerns of the Navigation system, we prioritize them based on the functional requirements as follows. Their relative importance is shown in order.

Table 4: Prioritizing Design Considerations for Navigation Subsystem

Name	Suitable Architecture
Solution to problem is heuristic	Blackboard
Opportunistic reasoning required	Blackboard
Incremental development required	Blackboard
Likely to change processing approaches	Layered
Likely to add processing approaches	Blackboard

The planning tasks are essentially based on heuristic rules. There are no absolute criteria to tell which path is better than the other. The whole activity involves a certain degree of uncertainty. Some algorithms do exist, but they are "rules of thumb" in nature. Achieving a good path plan requires participation of all "rules of thumb." A deterministic task schedule is unnecessary among those planning agents. Oopportunistic applications of knowledge are more likely needed. In most cases, it is impossible to derive a complete detailed path plan in advance because the DTE maps cannot provide detailed terrain features and the view of Perception is occasionally obstructed. Only a portion of situations surrounding the rover is actually "seen." Incrementally developing the path plan is required. Processing approaches are likely to be changed or added since this system is built for a research institution.

Judging from the table (either in terms of numbers or importance), we recommend that the blackboard architecture is a better design choice for the Navigation subsystem.

3.3 Integrating Navigation Subsystem with Rover System

There are four combinations of our architectural proposals:

1. Repository with Layered
2. Repository with Blackboard
3. Event-based with Layered
4. Event-based with Blackboard (our recommendation)

Because each of the Rover architectures was designed to operate with either of the Navigation components, there are few problems integrating each of the combinations. Our two candidate architectures for Navigation subsystem can readily fit into the two overall architectural designs for the Rover system. The Navigation subsystem is simply treated as a component in the overall Rover system. The interface is well defined. The Rover system passes goals and patterns to the Navigation subsystem. The Navigation subsystem returns the motion commands back to the Rover system. The Navigation subsystem will also interact with Rover system regarding information about the rover's current pose and terrain maps.

4   Adaptation for New Requirements

This section proposes three requirements changes for the rover system and four changes for the navigation system. The section also addresses the impact of those changes upon the design of both candidate architectures.

4.1   New Requirements for the Rover System

The first change adds the requirement that the rover support an additional kind of sensor, in this case, a laser rangefinder. The change has minimal effect on either the event-based or repository-based rover architectures.

1. Event - The design changes to implement this modification include the addition of hardware actuator interfaces for the rangefinder and signal interfaces coming from the rangefinder. Both interfaces are very similar to those provided for the black & white cameras. Because the changes only affect the navigation component, which also must be modified to use the new information, the changes are well encapsulated. There will be a performance price if the additional information is to be maintained by the Terrain Data Manager or if the information is to be transmitted to Earth.

2. Repository - Because either navigation component is designed to integrate within either of the rover architectures, the design changes to support this modification are similar to those required for the event-based system.

The second change adds a new mission type in which one rover "trails" another rover. Imagine a scenario where rover A is being teleoperated by a visitor to a science museum. Rover B would be operating using this new mission-type and be trailing after rover A in order to provide an external view of rover A's progress.

1. Repository - Because the various mission types are hard-coated into the Coordinator component, adding any new mission type requires a redesign and reconstruction of the Coordinator.

2. Event - Adapting the rover for this change requires a new component that would use imagery from the rover's video buffer in order to locate the other rover. Once the other rover has been located, the new component issues an autonomous mission goal to travel to a location near the other rover. When the other rover changes its position, the trailing rover's autonomous goal is recomputed and announced. This new component issues requests to the Video Interface in order to move the color camera. The addition of a new mission type also requires an additional set of event bindings.

The third proposed change adds the requirement that the rover keep a log or diary of the positions or locations that it visits. The log is eventually transmitted to Earth. The log has a maximum size - only the most recently visited points are recorded. The change has minimal effect on the event-based rover but has a significant effect on the repository-based rover.

1. Event - This change affects only the Event Manager component. The first modification is to add the mechanism which tracks events generated by the Position Estimation component. Other mechanisms are added to report the log's contents to Earth when requested.

2. Repository - Similar to the second change, this change requires significant changes to the Coordinator component because the control access rules need to be redesigned and reanalyzed. The system's state must be monitored for updates in the rover's current estimated position. Every update must be placed in the logfile.

4.2 New Requirements for the Navigation Subsystem

The first change adds the requirement that the navigation system support non-deterministic missions as a command directive in addition to the goal and pattern requirements. A mission command implementation would require the robot to redesign its intermediate goals on a conditional basis. In order to achieve a mission, the rover would have to navigate by following intelligent decisions until certain conditions are met. The conditions will be set according to the kind of experiment the rover is performing. For example, if the rover receives a mission where it is supposed to collect N samples of rocks of a given size and then return to its original position, the navigation system will probably have to generate random intermediate goals to explore the area and return to the original position when the condition of gathering N samples has been met by the experiment's goals.

1. Layered - In order to be able to apply this modifications to the proposed layered architecture, the system would have to embed new modules in the first layer parallel to the Pattern Planner which would check on the exit condition before generating a new intermediate goal. The exit condition would have to be received as part of the sensor data and be passed from the lowest layer to the highest layer.

2. Blackboard - The modifications to the Blackboard navigation system would involve adding new knowledge sources to the existing system and allowing them to receive input for the conditions set by the experiment running on board. Once the new mission knowledge sources update the corresponding structures on the blackboard, the rest of the system will follow execution the same way it used to.

Since this particular modification breaks the layered structure of the first proposed architecture by imposing that changes in the highest layer correspond directly to changes in the lowest layer, we can conclude that the blackboard system is better suited to realize this kind of functional modification. This is due to the flexibility of accommodating new knowledge sources into the existing system, and the system's property of allowing arbitrary modules to interact with each other, as opposed to the constraint determined by the layered architecture where a given layer can only communicate directly with its adjacent layers. This difference is a manifestation of the NAV-6 rule as explained in the Design Rules section 3.1, which favors the blackboard approach for the system since it is capable to easily add new processing approaches.

The second change adds the requirement that the navigation system support new methods for route planning. New searching algorithms could substitute the existing ones to generate more efficient paths. One such modification would substitute the energy concern with a requirement as a planning concern for traversability. This implies that the rover would be able to choose between energy and traversabillity of the terrain as a planning parameter

1. Blackboard - The blackboard system's adaptation would imply changing all the knowledge sources involved, and the difficulty arises in determining which of the knowledge sources intervene, in each of the planning stages.

2. Layered - The layered system would be capable of allowing such a change by changing the secrets of the implicated layers. If the energy/traversability concern is only taken into account in the pattern planning layer, then only the first layer would have to be modified.

In this case it would be easier to change the layered system since the information pertaining an algorithm is contained in one of the layers. This is due to the fact that layering can be used to group functionality hierarchically depending on what level of the problem they are solving. This is manifested in the NAV-5 rule which favors the layered proposal.

The third change adds the requirement that the navigation system support new data representations. One such modification would allow the global planner to calculate corridors instead of intermediate goals. This change would require the modules to process information as corridors instead of intermediate and immediate goals.

1. Layered - The layered system will require changes to be made to all the layers that deal with location data structures.

2. Blackboard - The knowledge sources would require redesign in order to handle the new data structures.

Neither of the two systems are adequately suited for dealing with changes of this nature.

The fourth change adds the requirement that the navigation system handle performance constraints. These constraint will not allow the security of the rover to decrease but would impose real-time deadlines to the navigation goals. In this situation the navigation system should be able to deliver goals in a deterministic amount of time and should be able to give up on the accuracy required.

1. Layered -

- One way to look at the problem would be to allow the layered system being able to time-out in each of the given layers and be able to relay a response to the next layer. This would produce many implementation problems since a simple time-out would not ensure reliable data being passed from one layer to another, and errors (such as unexecutable commands) would propagate.

- On the other hand, a layered system can provide us with a deterministic bounded computation time which can be tuned and adjusted to meet the required performance constraints.

2. Blackboard -

- The blackboard architecture operates structures on the blackboard layers that keep an updated version of the next step to be taken. In order to be able to meet timing constraints, the system would have to produce a heuristic solution in a bounded time, for example, in the form of a degraded solution. This is a more achievable goal than timing-out the processes of the layered system, and takes advantage of the fact that a solution of some level of feasibility is always available in a blackboard system.

- In a blackboard architecture, on the other hand, it would not be safe to take intermediate results of such a system since it would be very hard to determine their level of reliability. Also it is generally impossible to guarantee when a full solution can be derived.

5   References

[1] Simmons, Reid G., Structured Control for Autonomous Robots, IEEE Transactions on Robotics & Automation, 1993.

[2] David Garlan; Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds), *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company, 1993.

[3] H. Penny Nii. Blackboard Systems. *AI Maganizes* 7(3):38-53 and 7(4):82-107.

[4] Thomas G. Lane. A Design Space and Design Rules for User Interface Software Architecture. *Carnegie Mellon University Software Engineering Institute Technical Report CMU/SEI-90-TR-23.*